DIKU

Advanced Programming B1-2013

# Exam Report

Jonas Stig Kämpf Hansen

Total number of pages (including appendices, excluding frontpage): 56

7 November 2013

# Contents

# Introduction

This report has three sections, one for each of the parts of the exam. A print of all the code produced in answer to the questions, is attached as Appendix B. Some code fragments are also included in the report text, but otherwise there will be references to the appendix. The report is accompanied by source code for fully implemented, working solutions for all three questions in the file `Hansen.Jonas.src.zip`. The code was written using `Notepad++` on Windows, using UNIX-style line endings and ANSI-encoding.

# 1 Parser for Salsa

This implementation uses `SimpleParse` for parsing. As the supplied grammar does not immediately satisfy the requirements for using `SimpleParse`, we need to rewrite the grammar.

## 1.1 Grammar

We break the transformation of the grammar down into isolated steps. Generally, the grammar needs to be transformed into a grammar that:

- enforces operator precedences;
- is not left-recursive; and
- has been left factorised where required.

The rules are transformed according to the methods described in the course syllabus[1]. In the following, it is only briefly stated which principle leads to the rule(s) transformations, and, for the interesting cases, rewritten rule(s) are shown. Only rules requiring transformation are shown.

### DefComs

The rules for *DefComs* need to be left factorised. However, after the re-write, we need to ensure that *Program* requires there be at least one *DefCom*. This can be done in at least two ways: Rewrite *Program* to have *DefCom DefComs*, or introduce an additional rule for this purpose (*DefComs1*). In either case, the transformation is straightforward (and is shown in Appendix A, Code piece 11). When we implement this part of the grammar, we can simply use the `many1` parser combinator (and forego the *Defcoms1* rule).

### Command

The rules for *Command* have none of the required three properties for the grammar: Some are clearly left-recursive (alternatives 2 and 3); they do not enforce the precedence of `@` over `||`; and some have intersecting select sets, i.e. left factorisation is required.

The first insight is that the simplest forms of unambiguous commands are

> *CommandF* ::= `SIdents '->' Pos | '{' Command '}'`

Secondly, since `@` binds more tightly than `||`, we know that the left-hand side of an `'@'` can only be either a *CommandF* or another command of the form `Command '@' VIdent`. That is,

[1] "Grammars and parsing with Haskell Using Parser Combinators", draft version 2

`Command '||' Command '@' VIdent` should be parsed as `Command '||' (Command '@' VIdent)` per the precedence of `@`.

This leads to the following partially transformed grammar:

```
Command ::= SIdents '->' Pos
         | Command '@' VIdent
         | Command '||' Command
         | '{' Command '}'
```

Code piece 1: Original grammar

```
Command ::= Command '||' CommandA
         |  CommandA
CommandA ::= CommandA '@' VIdent
          |  CommandF
CommandF ::= SIdents '->' Pos
          |  '{' Command '}'
```

Code piece 2: Partial re-write

Note that this assumes that `||` is left-associative. This actually solves the left-factorisation requirement. However, we are not done, as the resulting grammar is (still) left-recursive. Transforming the grammar according to the rules in the syllabus results in:

```
Command ::= Command '||' CommandA
         |  CommandA
CommandA ::= CommandA '@' VIdent
          |  CommandF
CommandF ::= SIdents '->' Pos
          |  '{' Command '}'
```

Code piece 3: Partial re-write

```
Command ::= CommandA Copt
Copt    ::= '||' CommandA Copt
         |
CommandA ::= CommandF CAopt
CAopt    ::= '@' VIdent CAopt
          |
CommandF ::= SIdents '->' Pos
          |  '{' Command '}'
```

Code piece 4: Re-written grammar

### VIdents and SIdents

The rules for *VIdents* and *SIdents* need to be left-factorised. The transformation is pretty straightforward. Actually, we have a parser combinator, `chainl1`, which can abstract away this left-factorisation without explicitly putting in the new rules in our parser. A transformed grammar is not shown here, but can be found in Appendix A, Code piece 11.

### Expr and Prim

It is clear from the rules that `'+'` and `'-'` are left-associative. The transformation does not change that, but it does no longer follow from the rules themselves. However this is enforced by the way the parser will be implemented. Nothing has been stated with respect to precedence, however it is assumed that the two operators have the same precedence.

With *Prim* we see that two of the rules have the same non-terminal at the beginning of the rule. We need to perform a simple left-factorisation. We transform the rules for *Expr* and *Prim* as follows:

```
Expr ::= Prim
      |  Expr '+' Prim
      |  Expr '-' Prim
Prim ::= integer
      |  SIdent '.' 'x'
      |  SIdent '.' 'y'
      | '(' Expr ')'
```

Code piece 5: Original grammar

```
Expr ::= Prim Eopt
Eopt ::= '+' Prim Eopt
      |  '-' Prim Eopt
      |
Prim ::= integer
      |  SIdent '.' Dim
      | '(' Expr ')'
Dim ::= 'x' | 'y'
```

Code piece 6: Re-written grammar

### Grammar requirements

The transformed grammar (shown in Appendix A, Code piece 11) fulfils the grammar requirements from Figure 6 of the syllabus[2]:

- The rules for *Program*, *DefComs1*, *Command*, *CommandA*, *VIdents*, *SIdents* and *Expr* are of Form 0.
- The rule for *DefCom* is of Form 1. It is not evidently clear that this rule satisfies the requirement of disjoint *first* sets, i.e. we must have $First(Definition) \cap First(Command) = \emptyset$. $First(Definition)$ is the set of the terminals at the beginning of each of the rule alternatives. The *first* set of *Command* is $First(CommandA) = First(CommandF) = First(SIdents) \cup \{'\{'\} = First(SIdent) \cup \{'\{'\}$. Now, $First(SIdent)$ consists of strings of letters, digits and underscores, beginning with a lower case letter. However, such strings can precisely not be one of $First(Definition)$, so we do have that $First(Definition) \cap First(Command) = \emptyset$.

  The rule for *CommandF* is of Form 1, and satisfies the requirements, since `First('{' Command '}')` $= \{`\{`\}$ and `First(SIdents '->' Pos) = First(SIdent)`, which is some string starting with a lower case letter.

  The rule for *Prim* is of Form 1, and satisfies the requirements, since the $first$ sets of the alternatives are an integer, some string starting with a lower case letter and `'('`, respectively.

  The rules for *Definition*, *Pos*, *Dim* and *Colour* are also of Form 1, but are easily seen to meet the two requirements as each alternative for each rule starts with different terminals, thus the *first* sets are disjoint in all cases.

  For all rules of Form 1, it can be easily checked that no alternative can derive the empty string; for all but *DefCom* each alternative contains some terminal.
- The remaining rules, *DefComs*, *Copt*, *CAopt*, *VIopt*, *SIopt* and *Eopt*, are all of form 2. All but *Eopt* contain only one alternative $f_i$ besides the empty string $\epsilon$. However, clearly the $first$ sets in *Eopt* are disjoint.

  We then need to verify that the $follow$ sets do not intersect with the respective $first$ sets. This is easily seen for *DefComs* (since $Follow(DefComs) = \texttt{eof}$). The remaining $follow$ sets are described in Appendix A, Table A, and it follows that the requirement is met for each rule.

## 1.2 Relevant Design Decisions

The grammar transformation, see Section 1.1, was the most tricky part of this task. The design of the parser itself follows very closely from the guidelines in the syllabus[3]. The source code can be seen in Appendix B, Code piece 12.

`SimpleParse` was used for a parser combinator module, so parsing fails without specific error messages. Thus, parser errors are represented by the `SalsaParser.Error` datatype, which consists of a single unparameterised data constructor, `Error`.

## 1.3 Testing

### Valid programs

Since the transformed grammar is unambiguous (with our assumptions about the associativity of `'+'`, `'-'` and `'||'`), parsing programs is deterministic, i.e. we have that for every

[2] "Grammars and parsing with Haskell Using Parser Combinators", draft version 2
[3] "Grammars and parsing with Haskell Using Parser Combinators", draft version 2

source program, $p$, if parsing $p$ results in an $AST$ and another parsing of $p$ results in $AST'$, then $AST = AST'$. Note that this is *not* a one-to-one relation; for every $AST$ there can be (infinitely) many different source programs resulting in that exactly $AST$ (in particular we can have arbitrary whitespaces and/or parenthesis). However, if we translate the AST onto (any) *one* of these possible source programs, say into program $p_i$, we should get the same AST back, if we parse such source program $p_i$ using our parser. This is a very nice property that we would like to have for our parser, and one which is a good candidate for random testing:

***Property 1:*** *Given an arbitrary AST, a translation of this AST into a corresponding program source should parse into the same AST.*

We will test this property using QuickCheck to generate random ASTs and *a* corresponding source program. The test module is shown in Appendix B, Code piece 13. Testing has been done a few times for 10,000 test cases each. They all succeed.

Generally, we produce expressions according to the grammar. The test data generators are almost exact translations of the original grammar such that for each data structure we generate the necessary sub-data structures and at the same time make a local string representation of the data structure (inserting whitespace where required). Special care is needed for cases, where associativity and/or precedence matters. In these cases, we need to ensure that the created source text does actually correspond to the generated AST.

In test data generation, we impose some restrictions on the possible source programs being generated, due mainly to efficiency of running tests. Also, we do not randomly add parenthesis, unless these are strictly required by the chosen constructs (for instance, when a parenthesised command was chosen by the test data generator).

We could extend our testing of Property 1 by introducing random whitespaces and/or parenthesis into our source text, when generating it. However, this significantly slows down random testing, so testing is done with just a (fixed) single whitespace character where whitespaces are required or possible, and no additional parenthesis (note, though, that parenthesised *Command*s and *Expr*s may occur whenever such parenthesised constructs are selected by the test data generators).

Testing can be run by invoking `runTests` in module `SalsaParserTest`, or invoking `runManyTests` for runs of 10,000 QuickCheck tests.

### Invalid programs

The described random testing only tests valid programs. In practice, we would also like our parser to reject programs that do not adhere to the grammar. It would be easy to just create some random[4] gibberish and have the parser reject this, however, this will really not give us any confidence about the correctness of the parser.

A lot of cases of invalid programs will simply be rejected by the type safety of our AST data type: We cannot build an AST without at least the required elements of the proper type; and superfluous elements will be rejected, leading to a parsing error, unless they somehow can be used for the next structure (in which case the program may be valid anyway).

Consequently, the most interesting cases are the ones, where the types of the AST could potentially accept some construct that is actually not a valid program. For `SalsaAST`s this is the data types containing lists (including strings, but this is not directly testable as such, since the parser will ignore empty strings as whitespace); such lists could be empty and the AST would be valid type-wise, but (in all cases) invalid as a `Salsa` program.

Unit tests cover these three cases. We also unit test some additional test cases to make sure the parser distinguishes correctly between *SIdent*s, and *VIdent*s. We could also test that

[4]Well, not random, because we would need to ensure that the program is not actually grammatically correct, and then it would not be simple any more.

reserved words (keywords and colours) are not falsely accepted, but for this, we instead rely on the hard-coding and internal data structures of the parser being correct (with respect to spelling, look-ups, membership testing, etc.).

## 1.4 Assessment of Solution

In general, all parts of the parser (i.e. the complete grammar) has been implemented. `SalsaParser.hs` compiles and yields no warnings (with some type signatures added to the version of `SimpleParse` available on the course webpage) when compiled with

```
ghc -Wall -fno-warn-unused-do-bind SalsaParser.hs.
```

Further, `hlint` offers no suggestions.

Random testing using QuickCheck shows no parsing errors for up to 10,000 test cases in one run. The probability distribution for selecting between language constructs in the random generation of ASTs is not normal[5]. But no case has a higher probability ratio than 10-to-1, so all AST constructs should be included in sufficiently large test runs, including test runs of 10,000 test cases. I consider the lack of arbitrary whitespaces in test inputs to have no effect on the assessment of the testing results.

Also, unit tests for selected almost-legal programs show that the parser does not accept programs (and build ASTs) where there could be a (type-wise) valid AST, but where such AST does not represent a valid program. (Grammatical) correctness for all other cases is implied by type safety with respect to the AST.

In the future, the parser could be improved in at least one area. As it is, no specific error messages are supplied when parsing fails. This is mostly due the lack of support for such error messages in `SimpleParse`.

Hardly any internal data structures are used, and then only on relatively small data sets, so the performance of the parser is deemed to be dominated by the choice of parser module. However, no performance testing has been conducted as performance of the parser was adjudged to be secondary to correctness for this exam.

# 2 Interpreter for Salsa

## 2.1 Relevant Design Decisions

### 2.1.1 Semantics of Salsa

Most of the semantics of Salsa definitions and commands appear straightforward. With respect to associativity we see that it will not semantically matter for `||`, since two parallel commands cannot manipulate the same shapes. The arithmetic operators `+` and `-` are interpreted in the usual way. However, with respect to the `@` operator, we note that a command of the form `a b -> (1, 1) @ A @ B` must be parsed according the grammar like this: `(a b -> (1, 1) @ A) @ B`, which in a semantic sense can be stated as `com1 @ B`, where `com1 = a b -> @ A`. However, the semantics of such a command is not entirely clear. In particular, the effect of executing `com1` with `B` as the active view is unclear.

The semantic definition in the assignment states that *com* @ $Id$ temporarily makes $Id$ the active view for *com*. However, a *com* of the form $com_0$ @ $Id_0$ has no (other) effect than executing $com_0$ with $Id_0$ as the active form. In particular, it has no implications for any

[5]To lower the risk of diverging test cases, atomic constructs, save for parenthesised constructs, have a much higher probability.

other views. Consequently, we interpret statements of the form $com_0$ @ $Id_0$ @ $Id$ as being functionally equivalent to $com_0$ @ $Id_0$, i.e. we simply ignore any chained '@ $Id_i$'.[6]

### 2.1.2 Types and concerns

Salsa commands are executed in a context and may change (parts of) that context, but will mostly not have any (other) meaningful return value. Commands may have the side-effect of producing key frames, but, as elaborated below in Section 2.1.3, we shall consider that the concern of someone else. Consequently, we will need a type and monad instance defined as such:

```haskell
newtype SalsaCommand a = SalsaCommand { runSC :: Context -> (a, Context) }
instance Monad SalsaCommand where
  return x  = SalsaCommand $ \c -> (x, c)
  m >>= f   = SalsaCommand $ \c@(Context env _) ->
                let (x, Context _ shapes1) = runSC m c
                in runSC (f x) (Context env shapes1)
```

Code piece 7: Definition of `SalsaCommand`

Note that while running a `SalsaCommand` on some value of type `Context` may produce an altered value of type `Context`, but we throw away any alterations of the environment part of the specific context. The type for `Context` is:

```haskell
type Environment = (Defs, ActiveViews, FrameRate)
type Shapes = Map Ident ShapePos
data Context = Context Environment Shapes
```

Code piece 8: Data type `Context`

Type `Environment` corresponds to part (1) of the specification of contexts in the exam text, and type `Shapes` to part (2). Both these types contain some nested data structures to keep the appropriate data. For such nested structures with a variable number of entries, we use sets (used when we usually only perform operations on the entire set, set comparisons are important and/or where duplicate elimination is helpful, e.g. sets of definitions, set of active views, etc.) and maps (where data is often only accessed in parts using some key as filter, e.g. the positions of a shape in all views are keyed by the `sid` of that shape). Details are shown in Appendix B, Code piece 14.

Unlike for `SalsaCommand`s, something run as a `Salsa` monad *should* be allowed to alter the entire context and pass it down the chain. Thus, for type `Salsa`, we have:

```haskell
newtype Salsa a = Salsa { runS :: Context -> (a, Context) }
instance Monad Salsa where
  return x  = Salsa $ \c -> (x, c)
  m >>= f   = Salsa $ \c ->
                let (x, c') = runS m c
                in runS (f x) c'
```

Code piece 9: Definition of `Salsa`

Sometimes we need to evaluate some value of type `SalsaCommand` as a `Salsa`. But we need to ensure that such value still cannot modify the environment part of the context. We archive this by the function `liftC`:

[6] You could argue that such commands should in instead be understood to compound all $Id$s following the first @ such that one will not have to define a group to do an at-command in more than one non-active view. However, the above interpretation is much more simple to implement as it only requires that we consider the inner most @.

```
liftC :: SalsaCommand a -> Salsa a
liftC (SalsaCommand sc) = Salsa $ \c@(Context env _) ->
  let (x, Context _ ss) = sc c
  in (x, Context env ss)
```

Code piece 10: Definition of `liftC`

These type definitions, together with various functions for manipulating the context, make it very simple to define the interpreter functions `command`, `expr`, `definition` and `defCom`.

### 2.1.3 Producing animations

The task of `runProg` is to execute the `DefComs` of a `Program` in sequence, and return the resulting `Animation`. Since only commands produce key frames (save for the initial definitions before the first command, which also is a key frame), one might argue that the production of the frames of the animation should be the concern of `command` (since it is responsible for executing commands).

However, executing a command does not necessarily by itself define all the shapes of the new key frame; we still need to draw all the (other) shapes as well. Now, this appears to be outside the concern of `command` and more within the concern of whomever gave the command to `command`. Consequently, in this implementation, the production of animation frames (and the final output `Animation`) is done by the function `run` defined in `runProg`. This helper function takes the list of `DefComs` and executes them one by one. If the `DefCom` is a command, key frames are captured and animation instructions for the transition between key frames are produced. The concatenated animation is then returned.

### 2.1.4 Key frames and calculating frames

By the examples given, we can derive that a key frame is defined by the positions of shapes as they (i) were in the previous key frame, (ii) are affected by running a command or (iii) follow from definitions that may follow the command (up to, but not including, the next command). So there will be exactly one more key frame in an animation than the number of commands. The first frame is a special case in that the first key frame is defined by the start of the program, and all definitions that follow up to, but not including, the first command.

So when we reach a command in the list of `DefComs`, we can only really draw the animation between the previous two key frames. Consider the following sequence:

```
    | F_i ->       <-| F_j ->       <-| F_k ->
... | cmd_i [ defs ] | cmd_j [ defs ] | cmd_k ...
```

Upon reaching `cmd_k`, key frames $F_i$ and $F_j$ are now fully defined (but $F_k$ is not, unless it is the end of the program, in which case $F_j$ is the final key frame) and can be animated. The cases where `cmd_i` and/or `cmd_k` are, respectively, the start and/or end of the program are special cases, but works almost the same. Animation draws everything defined up to the point where $F_k$ begins.

We animate by first interpolating all shapes' view-positions (a shape's position within a view) from the positions of $F_i$ to $F_j$ and then translate those positions into lists of `GpxInstr`. Shapes defined after `cmd_j`, but before `cmd_k` appear at precisely the last frame of the animation from $F_i$ to $F_j$. Key frames can be drawn by the contents of the mutable part of the context (part (2) of the context specification from the exam text) with some lookups in the environment part to get the specific characteristics of the shapes.

The function `animate` with type signature

```
animate :: Integer -> Shapes -> Shapes -> ShapeDefs -> [Frame]
```

takes the frame rate, the positions of shapes in the various views in the two key frames, and some shape definitions, and then constructs the animation (frames) from the starting key frame to the end key frame. `animate` is heavily commented in the source code (Appendix B, Code piece 14), in part to argue the correctness of the algorithm.

#### 2.1.5 Errors

In general, per the exam specification, we assume that programs are error-free with respect to statically checkable errors. Consequently, we do not need to concern ourselves overly much with error checking and/or handling during interpretation. Thus, while some errors are caught (mainly because it is meaningless to continue) leading to an exception being thrown, some errors will not be caught and will just go silent.

### 2.2 Testing

Different test strategies are used for the various parts of the interpreter. All tests are included in the module `SalsaInterp`. The testing part of this module is shown in Appendix B, Code piece 15. All test pass.

- Random testing using QuickCheck is performed on the `interpolate` function. The following four properties are tested, where $n > 0$ is the input frame rate, $p_{start}$ and $p_{end}$ the start and end positions, respectively, and $ps$ the output list of positions (excluding the start position, which will then be $p_0$):
  - $\forall n, p_{start}, p_{end}.length(ps) = n$
  - $\forall n, p_{start}, p_{end}.(ps = [p_1, p_2, \ldots, p_n]) \Rightarrow (p_n = p_{end})$
  - $\forall n, p_{start}, p_{end}.ps = [(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)] \Rightarrow \big(\forall i.(x_i \leq x_{i+1}) \vee \forall i.(x_i \geq x_{i+1})\big) \wedge \big(\forall j.(y_j \leq y_{j+1}) \vee \forall j.(y_j \geq y_{j+1})\big)$, where $i, j$ ranges over numbers $1..(n-1)$.
  - $\forall n, p_{start}, p_{end}.ps = [p_1, p_2, \ldots, p_n] \Rightarrow \forall i.\mathsf{dist}(p_i, p_{i+1}) = \delta$, where $\mathsf{dist}(p_i, p_j)$ is the euclidean distance between points $p_i$ and $p_j$, and $\delta = \mathsf{dist}(p_{start}, p_{end})/n$. For this property, because of the rounding to whole pixels in the algorithm, we do not actually require strict equality, but just within rounding precision of $\delta$, which may be up to $2 \times 0.5$ (in the worst case $x_i$ was rounded down from .49 and $x_{i+1}$ was rounded up from .5, and similar for $y_i, y_{i+1}$); thus the distance only has to be within $\delta \pm 1.0$ for each axis. (We also have that $\Sigma_{i=0}^{n-1}\mathsf{dist}(p_i, p_{i+1}) = \mathsf{dist}(p_{start}, p_{end})$, where $p_0 = p_{start}$, but that is not enough for our purpose, since points may be unacceptably unevenly spaced.)
- Execution of commands and definitions by `command` and `definition`, respectively, is tested by unit testing of specific commands'/definitions' effect on the context. The evaluation of expressions by the function `expr` is also tested by (relatively few) unit tests for the interesting cases with respect to expressions. The concrete testing strategy is to have tests that cover each of the different cases (execution paths) of the respective functions, however we will only cover valid cases (those not resulting in an exception), as extensive error detection and handling is not part of the assignment.
- Generation of frames by `animate` is tested by random testing using QuickCheck. We want our `animate` to have at least the following properties, where $n > 0$ is the frame rate, $ss_0$ the (non-empty) set of shape positions in the start key frame, $ss_1$ the (non-empty) set of shape positions in the end key frame, $sds$ the shape definitions and $rs = [r_1, \ldots, r_n]$ the output list with frames $r_1, \ldots, r_n$, where $r_1$ is the first animated step after the starting key frame:

- $\forall n, ss_0, ss_1, sds.\mathsf{length}(rs) = n$.
- $\forall n.\forall i \in 1..n.\mathsf{length}(r_i) = \Sigma_{s \in S}(\mathsf{length}(s.views))$ instructions, where $S = \mathsf{shapes}(ss_0) = \mathsf{shapes}(ss_1)$ is the shapes in the sets of shape positions, and $s.views$ the list of $(v_i, pos_{vi})$ pairs for shape $s$' position in view $v_i$.
- Further, $\forall n.\forall s \in S.\forall i \in 1..n.r_i$ contains at least $\mathsf{length}(s.views)$ number of instructions for drawing a shape of $s$' type, dimensions and color. Because draw instructions are not uniquely identified, we can in principle have more shapes with the exact same dimensions and color be drawn on the same view in the same position. So we cannot tighten this requirement to exact amount.

Tests can be run by loading module `SalsaInterp` and invoking either `runInterpolateTests` , `runHUTests` or `runAnimateTests` for testing any one of the three test strategies described above, `runTests` to run them all. Fair warning: The default is set to quite a few number of test cases.

## 2.3 Assessment of Solution

All parts of `Salsa` have been implemented and are working. `SalsaInterp` compiles with the same flags as for `SalsaParser` without warnings, or suggestions from `hlint`. As described in Section 2.2, random testing confirms several properties about the most algorithmic parts of the interpreter, `interpolate` and `animate`. Although testing does not explicitly confirm that the generated draw instructions are actually precisely the correct ones, the quantity and overall characteristics of the generated output is as it should be. Visual testing shows that output animations appear to work as intended, albeit this testing has only been conducted on relatively few cases. We could do some more testing on the characteristics of the output animation, but I simply did not have the time to do it.

The possible effects of commands and/or definitions on the context for the animation are not that many, and the performed unit tests confirm the expected behaviour. At least for execution of definitions, testing could be extended by automated random testing. The technique would be, not unlike testing of the `SalsaParser` (see Section 1.3) to generate random definitions and their isolated expected effect on the context (e.g. that shape $s$ is now defined in the environment), however some updates of the context depend upon the content of the context, so the full effect of a definition cannot be determined locally.

No real performance testing has been done. A crucial part of the interpreter with respect to performance has to be the `animate` function. Most other parts are sequential steps that manipulate some mostly efficient data structure (e.g. maps, sets, small tuples). As part of random testing for correctness of `animate`, the function has been presented with key frames of up to 100 views and 200 shapes each with positions on some (random, but normalized around 33% the number of views) subsequence of the views. This is pretty serious work, and there is time to get your coffee before it finishes.

It is my understanding that the implementation in general adheres to monadic design principles as witnessed by the concise implementations of `command`, `definition` (with the context accessors/mutators) and to a lesser extent `runProg`. There may be the opportunity for some small tweaks here and there, but overall this is a reasonable result within the given time frame.

# 3 Atomic Transaction Server in Erlang

## 3.1 Relevant Design Decisions

### 3.1.1 Use of OTP

My implementation does not use OTP, but implements the server from the ground up. The reason for this is that it appeared that we would use only basic functionality of the generic servers in OTP (whether it be `gen_server` or `gen_fsm`), so I adjudged that the implementation would be simpler not having to abide strictly by the interfaces of those modules. Although this approach may in some cases lead to exposure of "ugly" boiler-plate code that could have been hiding by the generic modules, it simply makes it easier for me to reason about what is going on.

As stated in the exam text, we use `make_ref` to make the transaction references. But according to its documentation, it is only *almost* unique. However, for our purposes (keeping track of active transactions), we shall consider it unique.

### 3.1.2 Server design

The server keeps a list of transactions containing tuples of the transaction reference and the transaction process id: `{Ref, TID}`. The invariant is that all transactions in the list have as starting state the server's current state (or else they cannot, potentially, commit at a later time anyway). To keep this invariant, the server may, when a transaction successfully commits, abort all other transactions by simple emptying the list of transactions (and telling the transaction processes to stop). This behaviour also ensures that a successfully committed transaction will in any future (misguided) attempts to query it, etc., report as an aborted transaction.

This is also ensures that queries, updates and commits on aborted transactions always (and immediately) simple return `aborted`, namely if we cannot find the transaction in the transaction list (either because it has been aborted, because it never existed or because it has failed in some way). However, this will have the effect that a lookup for an aborted transaction will always traverse the entire list of (active) transactions. See further reflections on this later in the assessment of the solution in Section 3.2 below.

The code for the server is the main receive loop with some helper functions to perform safe queries, and start and keep track of transactions. Code for the `at_server` is shown in Appendix B, Code piece 19.

### 3.1.3 Transactions

Transactions are initialised with their starting state and maintain their own internal (potentially) modified state until the transaction commits (or is aborted, which in practice means stopped, by the server).

It is stated that updates to transactions, using `update_t`, should not block, but return immediately. If the update causes an error, the transaction should be aborted. However, since the update is asynchronous, the update may potentially fail without the server knowing immediately. The transactions sends an asynchronous message to the server that it has to be aborted, but the server may already be treating some message for a query or a commit. Thus, if an update fails in a transaction, the transaction shifts to a state of failure, such that all queries are replied to with `error`, and basically it just waits to be stopped by the server, which should happen, when the server processes the failure callback-message.

No messages about failed updates are sent to the client. Consequently, if the client wishes to confirm the status of the transaction, it will have to either send a query to it, or try to commit it.

It should also be noted that while updates are not-blocking, a hanging update function will block the transaction, and in particular can make the server hang, if that transaction is queried, since the servers query of the transaction *is* blocking. One could mitigate this by adding a timeout to the server's queries of transactions, and/or transactions processes' running of update functions.

### 3.1.4 Commits

It is not stated explicitly, but it follows from the return value of `commit_t` that this call should be blocking. The question, however, is for how long it should be blocking. Basically, there are two options:

Either the server makes sure that the transaction is scheduled for committing (i.e. blocks all other transactions from committing), but the actual commit depends upon the transaction process sending back the state. In this case the server could potentially return `ok` even though the transaction process has crashed thereby never actually sending back the new state. Alternatively, the server must, after having received the client's request for a commit, itself *block* until the transaction process replies with its state. This can potentially cause the server to block for some time, but at least the server state will not be inconsistent with its outwards appearance, and queries will not be replied erroneously to in the mean time.

### 3.1.5 Extended API

The extended API functions (code for which is shown in Appendix B, Code piece 20) can be implemented as such:

- To force an abort of a transaction, we simply update (or query) the transaction with some function that always fails, e.g. by throwing an exception. This forces the server to abort the transaction per the API specification.
- To catch whether the update part of `tryUpdate/2` succeeds, we wrap the updating function `Fun` in an error handler that sends a message back to the client with a word on the success or failure of the updating function. The wrapper's only function is this callback; it passes the result of `Fun(State)` on to the transaction process, or throws an exception, if `Fun(State)` failed.
- I understand the specification of `ensureUpdate/2` such that the function will try updating the AT's state even though some other transaction may have updated the state compared to when `ensureUpdate/2` was originally called, i.e. it keeps reapplying the update until the update function fails or the update succeeds. Thus, the supplied `Fun` may potentially be applied to a number of different states before maybe succeeding. Then the implementation of this function is almost the same as for `tryUpdate/2`, only that if `tryUpdate/2` return `aborted`, we recursively calls `ensureUpdate/2` to keep trying until either the transaction successfully updates and commits, or the update fails.
- With respect to `choiceUpdate/3`, I am not entirely sure what is meant by trying each update 'in parallel' for each of the elements in the list. No matter how we do it, we have to extract the arguments from the list and this cannot be done completely in parallel; not even if we divide and conquer (for instance, if we spawn $n$ new processes each responsible for one argument of the list). We still have to spawn the processes sequentially (or at least logarithmically), so some arguments will be favoured with a

"head-start". Thus, I have chosen to understand this requirement merely to mean that we do not wait for the result of one update before trying the next. So basically we spawn $n$ new processes that each tries to update (using `tryUpdate/2`, and then callback the client process with the result. Only the first such callback message is catched by the client, so actually we have the first to callback wins (not the first to actually *update*).

Note that the `choiceUpdate/3` blocks until at least one process reports a commit. This could cause `choiceUpdate/3` to hang. Therefore, I have implemented an additional `choiceUpdate/4` which also takes a timeout which will have the obvious implication that `choiceUpdate/4` aborts after the timeout.[7]

## 3.2 Assessment of Solution

### 3.2.1 Testing

All parts of the multi-process server are implemented, working and tested by the described unit tests. Both modules compile without any warnings, and do not produce any unnecessary debug information to the console.

EUnit has been used for both black-box testing of the API exposed by `at_server` and some white-box testing of the internal implementation helper functions. Test cases are included in the module, and can be run (when testing is enabled) by the normal EUnit method of `at_server:test()`. The same applies to the `at_extapi` module. Unit tests may also in this case be run with `at_extapi:test()`.

Tests have been designed to show the specified behaviour, and some other identified interesting border cases. Test results gives high confidence in the correctness of the implementation of the `at_server` with respect to the specified behaviour. Testing of the extending API is more tricky seeing as the behaviour of the extended API functions are defined by multiple agents interacting with the same server. The test strategy here was to set up some agents that have strategically placed 'stop lights' within their code paths so as to explicitly, and deterministically, control the interacting transactions. It is hard to reason about, and testing may be incomplete here for some potentially critical real-life scenarios. Random testing appears to out of the question here, since introducing randomness into these kinds of race-conditions seems to be not very 'checkable', and thereby useless as grounds for assessment of correctness.

### 3.2.2 Data structures

As mentioned in the design description, a lookup traverses a list of (active) transactions. Another data structure having constant time or logarithmic time lookups could easily be used instead, however it was considered not the primary focus of this assignment, which instead was on effectiveness of implementing and correctness before straight-up performance. No other performance concerns were readily identified.

[7] You could perhaps also have the worker processes send a message back if the update fails, and then have the parent process collect failures, and abort if all children report failures.

## A Grammar

Code piece 11: Re-written grammar

```
   Program ::= DefComs1
  DefComs1 ::= DefCom DefComs
   DefComs ::= DefCom DefComs
             |
    DefCom ::= Command
             |  Definition
Definition ::= 'viewdef' VIdent Expr Expr
             |  'rectangle' SIdent Expr Expr Expr Expr Colour
             |  'circle' SIdent Expr Expr Expr Colour
             |  'view' VIdent
             |  'group' VIdent '[' VIdents ']'
   Command ::= CommandA Copt
      Copt ::= '||' CommandA Copt
             |
  CommandA ::= CommandF CAopt
     CAopt ::= '@' VIdent CAopt
             |
  CommandF ::= SIdents '->' Pos
             |  '{' Command '}'
   VIdents ::= VIdent VIopt
     VIopt ::= VIdent VIopt
             |
   SIdents ::= SIdent SIopt
     SIopt ::= SIdent SIopt
             |
       Pos ::= '(' Expr ',' Expr ')'
             |  '+' '(' Expr ',' Expr ')'
      Expr ::= Prim Eopt
      Eopt ::= '+' Prim Eopt
             |  '-' Prim Eopt
             |
      Prim ::= integer
             |  SIdent '.' Dim
             | '(' Expr ')'
       Dim ::= 'x' | 'y'
    Colour ::= 'blue'  'plum' | 'red' | 'green' | 'orange'
```

Equations to solve with respect to smallest set of terminals (leaving out the trivially solveable equations of form $Follow(T) \supseteq Follow(T)$):

$$\begin{aligned}
Follow(Copt) &\supseteq Follow(Command) \\
&\supseteq First(Command) \cup First(Definition) \cup \{\mathsf{eof}\} \\
&= First(SIdent) \cup \{\text{`\{'}, \mathsf{eof}\} \cup First(Definition) \\
First(\text{`||`}) &= \{\text{`||`}\}
\end{aligned}$$

Which gives us: $Follow(Copt) \cap First(\text{`||`}) = \{\}$

$$\begin{aligned}
Follow(CAopt) &\supseteq Follow(CommandA) \\
&\supseteq First(Copt) \cup Follow(Copt) \\
&= \{\text{`||`}\} \cup Follow(Copt) \\
First(\text{`@`}) &= \{\text{`@`}\}
\end{aligned}$$

Which gives us: $Follow(CAopt) \cap First(\text{`@`}) = \{\}$

$$Follow(VIopt) \supseteq Follow(VIdents) \supseteq First(\text{`]`})$$

Which gives us: $Follow(VIopt) \cap First(VIdent) = \{\}$

$$Follow(SIopt) \supseteq Follow(SIdents) \supseteq First(\text{`}- >\text{`})$$

Which gives us: $Follow(SIopt) \cap First(SIdent) = \{\}$

$$\begin{aligned}
Follow(Eopt) &\supseteq Follow(Expr) \\
&\supseteq First(Expr) \cup First(\text{`,`}) \cup First(\text{`)`}) \cup First(Colour) \cup \\
&\quad Follow(Definition) \\
&= First(Prim) \cup \{\text{`,`}, \text{`)`}\} \cup First(Colour) \cup Follow(Definition) \\
&= First(Prim) \cup \{\text{`,`}, \text{`)`}\} \cup First(Colour) \cup \\
&\quad First(Definition) \cup First(CommandF)
\end{aligned}$$

Which gives us: $Follow(Eopt) \cap First(\text{`}+\text{`}, \text{`}-\text{`}) = \{\}$

The sets $First(SIdent)$, $First(VIdent)$ and $First(Definition)$ are not written out, but obviously contain the terminals defined by the grammar rules.

Table 1: $Follow$ and $First(f_i)$ sets for type 2 rules in transformed grammar

## B Source code

Code piece 12: Module SalsaParser

```haskell
{-
  Parser for Salsa

  Based on:
  Skeleton for Salsa parser
  To be used at the exam for Advanced Programming, B1-2013

  Student:  Jonas Stig Kaempf Hansen
  KU-id:

-}

module SalsaParser
  ( Error
  , parseString
  , parseFile
  ) where

import Data.Char
import Control.Monad(liftM4)
import qualified Data.Map as Map
import SimpleParse

import SalsaAst

{--------------
  Parsing rules for grammar
--------------}

program :: Parser Program
program = defcoms

defcoms :: Parser [DefCom]
defcoms = many1 defcom

defcom :: Parser DefCom
defcom = (do d <- definition
             return $ Def d)
       <|> (do c <- command
               return $ Com c)

definition :: Parser Definition
definition = viewdef <|> rectangle <|> circle <|> view <|> group
  where
    viewdef =
       do symbol "viewdef"
          vid <- vident
          e1 <- expr
          e2 <- expr
          return $ Viewdef vid e1 e2
    fig = liftM4 (,,,) sident expr expr expr
    rectangle =
       do symbol "rectangle"
          (sid,e1,e2,e3) <- fig
          e4 <- expr
          c <- colour
          return $ Rectangle sid e1 e2 e3 e4 c
    circle =
       do symbol "circle"
          (sid,e1,e2,e3) <- fig
          c <- colour
          return $ Circle sid e1 e2 e3 c
    view =
       do symbol "view"
          vid <- vident
          return $ View vid
    group =
       do symbol "group"
          vid <- vident
          symbol "["
          vids <- vidents
          symbol "]"
          return $ Group vid vids

command :: Parser Command
command = commandA `chainl1` connectop
        where connectop = do symbol "||"
                             return Par

commandA :: Parser Command
commandA = do f <- commandF
              cAopt f

cAopt :: Command -> Parser Command
cAopt c = (do symbol "@"
              vid <- vident
              cAopt $ At c vid)
        <|> return c

commandF :: Parser Command
commandF = (do sids <- sidents
               symbol "->"
               p <- pos
               return $ Move sids p)
        <|> (do symbol "{"
                c <- command
                symbol "}"
                return c)

vidents :: Parser [Ident]
vidents = many1 vident

sidents :: Parser [Ident]
sidents = many1 sident

pos :: Parser Pos
pos = do s <- option (symbol "+")
         symbol "("
         e1 <- expr
         symbol ","
         e2 <- expr
         symbol ")"
         case s of
           Nothing -> return $ Abs e1 e2
           Just _  -> return $ Rel e1 e2

expr :: Parser Expr
expr = prim `chainl1` (plus <|> minus)
```

```haskell
     where plus = do symbol "+"
                     return Plus
           minus = do symbol "-"
                      return Minus

prim :: Parser Expr
prim = constn <|> proj <|> parExpr
     where constn = do n <- integer
                       return $ Const n
           proj = do sid <- sident
                     symbol "."
                     dim sid
           parExpr = do symbol "("
                        e <- expr
                        symbol ")"
                        return e

dim :: Ident -> Parser Expr
dim sid = opt "x" Xproj <|> opt "y" Yproj
         where opt s t = do symbol s
                            return $ t sid

colour :: Parser Colour
colour = token (do name <- many1 (satisfy isLetter)
                   case Map.lookup name colors of
                     Just c -> return c
                     Nothing -> reject)

{--------------
  Reserved symbols
--------------}

keywords :: [String]
keywords = ["viewdef", "rectangle", "circle", "group", "view"]

colors :: Map.Map String Colour
colors = Map.fromList [("blue", Blue)
                      ,("plum", Plum)
                      ,("red", Red)
                      ,("green", Green)
                      ,("orange", Orange)
                      ]

reserved :: [String]
reserved = keywords ++ Map.keys colors

{--------------
  Parsers for identifiers and numbers
  [Losely based on my hand-in for assignment 1]
--------------}

vident :: Parser Ident
vident = ident isUpper

sident :: Parser Ident
sident = ident isLower

-- Get identifier begining with symbol satisfying predicate p
ident :: (Char -> Bool) -> Parser Ident
ident p = token (do first <- satisfy p
                    rest <- many (letter <|> num <|> undsc)
                    let name = first:rest in
                      if name `notElem` reserved then return name
                      else reject)
          where letter = satisfy isLetter
                num = satisfy isDigit
                undsc = char '_'

integer :: Parser Integer
integer = token (do n <- digits
                    return $ read n)

digits :: Parser String
digits = many1 (satisfy isDigit)

{--------------
  Module API
--------------}

-- Type for parse errors
data Error = Error deriving (Show, Eq)

parseString :: String -> Either Error Program
parseString input = case par_result of
                      (p, ""):_ -> Right p
                      _ -> Left Error
                  where par_result = parse (do prog <- program
                                               token eof
                                               return prog) input

parseFile :: FilePath -> IO (Either Error Program)
parseFile filename = fmap parseString $ readFile filename
```

Code piece 13: Tests for `SalsaParser` in test module `SalsaParserTest`

```haskell
{-
  Random Testing using QuickCheck for SalsaParser
-}
module SalsaParserTest where

import qualified Test.QuickCheck as QC
import qualified Test.HUnit as HU

import qualified SalsaParser as SP
import SalsaAst

{-
  Generating random ASTs

  We generate the ASTs and its source representation at the same time. For each
  possible rule in the grammar, we generate a random concrete instance of such
  rule, which includes possible random sub-components (non-terminals) used in
  the rule), consisting of an AST representation and a string representation
  of that rule instance. These two representations are then compounded into
  the entire source and complete AST of the program, respectively.

-}

--------------
-- Whitespaces
-- NOT REALLY USED FOR THIS TESTING
--------------
whitespaces :: String
whitespaces = " \t\n"
newtype Whitespace = WS Char
instance QC.Arbitrary Whitespace where
  arbitrary = do c <- QC.elements whitespaces
                 return $ WS c

someBlanks :: QC.Gen String
someBlanks = do bs <- QC.arbitrary -- More spaces: bs <- QC.listOf QC.arbitrary
                return $ map unpack bs
           where unpack (WS c) = c

-- someBlanks1 :: QC.Gen String
-- someBlanks1 = do (WS b) <- QC.arbitrary
                 -- bs <- someBlanks
                 -- return (b:bs)
someBlanks1 :: QC.Gen String
someBlanks1 = return " "

-----------
-- Integers
-----------
newtype ConstInt = ConstInt Integer deriving (Eq, Show)
instance QC.Arbitrary ConstInt where
  arbitrary = do
    (QC.NonNegative n) <- QC.arbitrary
    return $ ConstInt n

---------
-- Idents
---------
videntStart :: String
videntStart = ['A'..'Z']
```

```
sidentStart :: String
sidentStart = ['a'..'z']
nameChars :: String
nameChars = sidentStart ++ videntStart ++ ['0'..'9'] ++ "_"

-- Max length of generated identifiers
identSize :: Int
identSize = 10

-- Max number of idents in lists of identifiers
idListMax :: Int
idListMax = 3

newtype TestVIdent = TestVIdent (String, String) deriving (Eq, Show)
instance QC.Arbitrary TestVIdent where
  arbitrary = do s <- getIdent videntStart
                 return $ TestVIdent (s, s)

newtype TestSIdent = TestSIdent (String, String) deriving (Eq, Show)
instance QC.Arbitrary TestSIdent where
  arbitrary = do s <- getIdent sidentStart
                 return $ TestSIdent (s, s)

getIdent :: String -> QC.Gen String
getIdent firstChar = do
  c <- QC.elements firstChar
  cs <- QC.vectorOf identSize $ QC.elements nameChars
  return (c:cs)

getVIdents :: QC.Gen [TestVIdent]
getVIdents = QC.vectorOf idListMax QC.arbitrary

getSIdents :: QC.Gen [TestSIdent]
getSIdents = QC.vectorOf idListMax QC.arbitrary

---------
-- Colour
---------
colors :: QC.Gen (String, Colour)
colors = QC.elements
         [("blue", Blue), ("plum", Plum),
          ("red", Red), ("green", Green),
          ("orange", Orange)]

--------------
-- Expressions
--------------
projs :: [(String, String -> Expr)]
projs = [("x", Xproj), ("y", Yproj)]
newtype TestPrim = TestPrim (String, Expr) deriving (Eq, Show)
instance QC.Arbitrary TestPrim where
  arbitrary = QC.frequency [(50, anInteger), (20, aProj), (5, aPExpr)]
    where
      anInteger = do
        (ConstInt n) <- QC.arbitrary
        return $ TestPrim (show n, Const n)
      aProj = do
        (TestSIdent (sid, _)) <- QC.arbitrary
        (dim, proj) <- QC.elements projs
        let s = sid ++ "." ++ dim
        return $ TestPrim (s, proj sid)
```

```haskell
      aPExpr = do
        (TestExpr (s, e)) <- QC.arbitrary
        let s' = "(" ++ s ++ ")"
        return $ TestPrim (s', e)

arithOps :: [(String, Expr -> Expr -> Expr)]
arithOps = [(" + ", Plus), (" - ", Minus)]
newtype TestExpr = TestExpr (String, Expr) deriving (Eq, Show)
instance QC.Arbitrary TestExpr where
  arbitrary = QC.frequency [(10, aPrim), (1, anArith)]
    where
      aPrim = do
        (TestPrim p) <- QC.arbitrary
        return $ TestExpr p
      anArith = do
        (TestExpr (s1, e)) <- QC.arbitrary
        (TestPrim (s2, p)) <- QC.arbitrary
        (sign, op) <- QC.elements arithOps
        let s = s1 ++ sign ++ s2
        return $ TestExpr (s, op e p)

------
-- Pos
------
posTypes :: [(String, Expr -> Expr -> Pos)]
posTypes = [("+", Rel), ("", Abs)]
newtype TestPos = TestPos (String, Pos) deriving (Eq, Show)
instance QC.Arbitrary TestPos where
  arbitrary = do
    (TestExpr (s1, e1)) <- QC.arbitrary
    (TestExpr (s2, e2)) <- QC.arbitrary
    (sign, con) <- QC.elements posTypes
    let s = sign ++ "(" ++ s1 ++ ", " ++ s2 ++ ")"
    return $ TestPos (s, con e1 e2)

--------------
-- Definitions
--------------
newtype TestDef = TestDef (String, Definition) deriving (Eq, Show)
instance QC.Arbitrary TestDef where
  arbitrary = QC.frequency [(1, aViewdef)
                           ,(1, aRect)
                           ,(1, aCir)
                           ,(1, aView)
                           ,(1, aGroup)]
    where
      aViewdef = do
        (TestVIdent (vid,_)) <- QC.arbitrary
        (TestExpr (s1, e1)) <- QC.arbitrary
        (TestExpr (s2, e2)) <- QC.arbitrary
        let s = unwords ["viewdef", vid, s1, s2]
        return $ TestDef (s, Viewdef vid e1 e2)
      fig = do
        (TestSIdent (sid,_)) <- QC.arbitrary
        (TestExpr (s1, e1)) <- QC.arbitrary
        (TestExpr (s2, e2)) <- QC.arbitrary
        (TestExpr (s3, e3)) <- QC.arbitrary
        return (sid, s1, e1, s2, e2, s3, e3)
      aRect = do
        (sid, s1, e1, s2, e2, s3, e3) <- fig
        (TestExpr (s4, e4)) <- QC.arbitrary
        (s5, c) <- colors
        let s = unwords ["rectangle", sid, s1, s2, s3, s4, s5]
        return $ TestDef (s, Rectangle sid e1 e2 e3 e4 c)
      aCir = do
        (sid, s1, e1, s2, e2, s3, e3) <- fig
        (s5, c) <- colors
        let s = unwords ["circle", sid, s1, s2, s3, s5]
        return $ TestDef (s, Circle sid e1 e2 e3 c)
      aView = do
        (TestVIdent (vid,_)) <- QC.arbitrary
        let s = unwords ["view", vid]
        return $ TestDef (s, View vid)
      aGroup = do
        (TestVIdent (vid,_)) <- QC.arbitrary
        vs <- getVIdents
        let vids = map unpack vs
        let ss = ["group", vid, "["] ++ vids ++ ["]"]
        let s = unwords ss
        return $ TestDef (s, Group vid vids)
      unpack (TestVIdent (vid, _)) = vid

-----------
-- Commands
-----------
newtype TestCom = TestCom (String, Command) deriving (Eq, Show)
instance QC.Arbitrary TestCom where
  arbitrary = QC.frequency [(10, aCommandA), (1, aParCommand)]
    where
      aCommandA = do
        (TestComA c) <- QC.arbitrary
        return $ TestCom c
      aParCommand = do
        (TestCom (s1, c1)) <- QC.arbitrary
        (TestComA (s2, c2)) <- QC.arbitrary
        let ss = unwords [s1, "||", s2]
        return $ TestCom (ss, Par c1 c2)

newtype TestComA = TestComA (String, Command) deriving (Eq, Show)
instance QC.Arbitrary TestComA where
  arbitrary = QC.frequency [(5, aCommandF), (1, anAtCommand)]
    where
      aCommandF = do
        (TestComF c) <- QC.arbitrary
        return $ TestComA c
      anAtCommand = do
        (TestComA (s1, c1)) <- QC.arbitrary
        (TestVIdent (vid, _)) <- QC.arbitrary
        let ss = unwords [s1, "@", vid]
        return $ TestComA (ss, At c1 vid)

newtype TestComF = TestComF (String, Command) deriving (Eq, Show)
instance QC.Arbitrary TestComF where
  arbitrary = QC.frequency [(10, aMove), (1, aPCommand)]
    where
      aMove = do
        sid_list <- getSIdents
        (TestPos (s_pos, pos)) <- QC.arbitrary
        let sids = map unpack sid_list
        let ss = sids ++ ["->", s_pos]
        let s = unwords ss
        return $ TestComF (s, Move sids pos)
      unpack (TestSIdent (sid, _)) = sid
      aPCommand = do
        (TestCom (s, c)) <- QC.arbitrary
        let ss = "{" ++ s ++ "}"
        return $ TestComF (ss, c)

----------
-- DefComs
----------
newtype TestDefCom = TestDefCom (String, DefCom) deriving (Eq, Show)
instance QC.Arbitrary TestDefCom where
  arbitrary = QC.frequency [(2, aDefinition), (1, aCommand)]
    where
      aDefinition = do
        (TestDef (s, d)) <- QC.arbitrary
        return $ TestDefCom (s, Def d)
      aCommand = do
        (TestCom (s, c)) <- QC.arbitrary
        return $ TestDefCom (s, Com c)

-----------
-- Programs
-----------
newtype TestProgram = TestProgram (String, Program)
  deriving (Eq, Show)

instance QC.Arbitrary TestProgram where
  arbitrary = do
    defComs <- QC.listOf1 QC.arbitrary
    let ts = map unpack defComs
    let ss = map fst ts
    let ds = map snd ts
    ptxt <- makeSource ss
    return $ TestProgram (ptxt, ds)

    where
      makeSource :: [String] -> QC.Gen String
      makeSource ss = return $ unwords ss
      -- makeSource [] = return ""
      -- makeSource (s:ss) = do
        -- bs <- someBlanks1
        -- rest <- makeSource ss
        -- return (s ++ bs ++ rest)

      unpack (TestDefCom sd) = sd

-------------
-- Properties
-------------

-- Property 1
prop_pDeterminism :: TestProgram -> Bool
prop_pDeterminism (TestProgram (s, ast)) =
  SP.parseString s == Right ast

-- Reject programs that type as ASTs, but are illegal according to the grammar
tEmptyLists :: [(String, String)]
tEmptyLists = [("No defcoms", "")
              ,("No SIdent in move", " -> (1,1)")
              ,("No VIdent in group", "group A [ ]")]
tWrongSIdent :: [(String, String)]
tWrongSIdent = [("Upper case", "rectangle Ax 1 1 1 1 blue")
               ,("Digit"     , "rectangle 9x 1 1 1 1 blue")
               ,("Underscore", "rectangle _x 1 1 1 1 blue")
               ,("Reserved 1", "rectangle view 1 1 1 1 blue")
               ,("Reserved 2", "rectangle plum 1 1 1 1 blue")]
tWrongVIdent :: [(String, String)]
tWrongVIdent = [("Lower case", "viewdef aB 1 1")
               ,("Digit"     , "viewdef 9B 1 1")
               ,("Underscore", "viewdef _B 1 1")]

pParseError :: Either SP.Error Program -> Bool
pParseError (Left _) = True
pParseError _        = False

tParseError :: (String, String) -> HU.Test
tParseError (tName, input) =
  HU.TestCase $ HU.assertBool tName $ pParseError $ SP.parseString input

testsHU :: HU.Test
testsHU = HU.TestList
  [HU.TestLabel "Non-empty lists" $ HU.TestList (map tParseError tEmptyLists)
  ,HU.TestLabel "Wrong SIdent"    $ HU.TestList (map tParseError tWrongSIdent)
  ,HU.TestLabel "Wrong VIdent"    $ HU.TestList (map tParseError tWrongVIdent)]

-- Done with test cases

runTests :: IO HU.Counts
runTests = do
  QC.quickCheck prop_pDeterminism
  HU.runTestTT testsHU

runManyTests :: IO ()
runManyTests = QC.quickCheckWith QC.stdArgs {QC.maxSuccess = 10000}
               prop_pDeterminism
```

Code piece 14: Module `SalsaInterp (excluding tests)`

```
--
-- Implementation of Salsa interpreter
--
-- By Jonas Stig Kaempf Hansen
-- To be used at the exam for Advanced Programming, B1-2013
--

module SalsaInterp
  (Position, interpolate, runProg, runTests)
where

-- Test dependencies. Test cases are at the end
import Test.HUnit
import Test.QuickCheck
  ( arbitrary, choose, vectorOf, oneof, elements
  , Gen, Arbitrary, quickCheckWith, stdArgs, maxSuccess)
import qualified Test.QuickCheck as QC

-- API implementation dependencies
import SalsaAst
import Gpx

import Control.Monad(liftM, liftM2, liftM3)
import Data.Maybe(fromMaybe)

import Data.Set (Set)
import qualified Data.Set as Set

import Data.Map (Map)
import qualified Data.Map as Map

import Data.List (sort, transpose)
import qualified Data.List as List

--
-- Interpolate move from p1 to pn in n steps
--
type Position = (Integer, Integer)
interpolate :: Integer -> Position -> Position -> [Position]
interpolate n p1 pn
  | n < 0  = error $ "Error! Tried to interpolate with n = " ++ show n
  | n == 0 = [] -- Specification says that list has n elements!
  | otherwise = [pos i | i <- steps]
  where
    (x1, y1) = p1
    (xn, yn) = pn
    (dx, dy) = (xn - x1, yn - y1) -- distance to move in total
    steps = [1..n]
        -- excludes the initial pos (p1) from steps;
        -- use [0..n] to include
    pos :: Integer -> Position
    pos i = (x1 + round dxi, y1 + round dyi)
      where
        n' :: Double
        n' = fromInteger n
        (dxi, dyi) = (fromInteger (i * dx) / n', fromInteger (i * dy) / n')

--
-- Types Context and SalsaCommand
--
```

```haskell
data Shape = Rect Integer Integer String
           | Cir  Integer String
     deriving (Show, Eq)
type ShapeDefs = Map Ident Shape
data ViewType = Vw Integer Integer | Gr [Ident] deriving (Show, Eq)
type ViewDefs = Map Ident ViewType
type Defs = (ViewDefs, ShapeDefs)

type ActiveViews = Set Ident
type FrameRate = Integer
type Environment = (Defs, ActiveViews, FrameRate)

type ShapePos = [(Ident, Position)] -- should properly be sets
type Shapes = Map Ident ShapePos
data Context = Context Environment Shapes deriving (Show, Eq)

newtype SalsaCommand a = SalsaCommand { runSC :: Context -> (a, Context) }
instance Monad SalsaCommand where
  return x  = SalsaCommand $ \c -> (x, c)
  m >>= f   = SalsaCommand $ \c@(Context env _) ->
                let (x, Context _ shapes1) = runSC m c
                in runSC (f x) (Context env shapes1)

--
-- functions for manipulating the context
--

-- * Context initializers

emptyEnvironment :: FrameRate -> Environment
emptyEnvironment n = (emptyDefs, Set.empty, n)
  where emptyDefs = (Map.empty, Map.empty)

emptyContext :: FrameRate -> Context
emptyContext n = Context (emptyEnvironment n) Map.empty

-- * Context lookups

askContext :: SalsaCommand Context
askContext = SalsaCommand $ \c -> (c, c)

askEnv :: SalsaCommand Environment
askEnv = do (Context env _) <- askContext
            return env

askShapes :: SalsaCommand Shapes
askShapes = do (Context _ ss) <- askContext
               return ss

lookupActive :: SalsaCommand ActiveViews
lookupActive = do (_, av, _) <- askEnv
                  return av

lookupShapePos :: Ident -> SalsaCommand (Maybe ShapePos)
lookupShapePos sid = do (Context _ ss) <- askContext
                        return $ Map.lookup sid ss

lookupViewDef :: Ident -> SalsaCommand (Maybe ViewType)
lookupViewDef vid = do vDefs <- getViewDefs
                       return $ Map.lookup vid vDefs
```

```haskell
lookupDefs :: Environment -> SalsaCommand Defs
lookupDefs (defs, _, _) = return defs

-- * Getters

getShapes :: SalsaCommand Shapes
getShapes = askShapes

getDefs :: SalsaCommand Defs
getDefs = lookupDefs =<< askEnv

getShapeDefs :: SalsaCommand ShapeDefs
getShapeDefs = liftM snd getDefs

getViewDefs :: SalsaCommand ViewDefs
getViewDefs = liftM fst getDefs

getActive :: SalsaCommand [Ident]
getActive = do av <- lookupActive
               return $ Set.toList av

-- * Local context mutation

-- Allows for local changes to context, but any changes will simply
-- be disregarded after the SalsaCommand has executed
local :: (Context -> Context) -> SalsaCommand a -> SalsaCommand a
local f m = SalsaCommand $ \c -> let c' = f c
                                 in runSC m c'


-- * Context mutators

insertShapeDef :: Ident -> Shape -> Environment -> Environment
insertShapeDef sid shape ((vDefs, sDefs), a, f) =
  ((vDefs, Map.insert sid shape sDefs), a, f)

insertShapePos :: Ident -> ShapePos -> Shapes -> Shapes
insertShapePos = Map.insert

insertViewDef :: Ident -> ViewType -> Environment -> Environment
insertViewDef vid view ((vDefs, sDefs), a, f) =
  ((Map.insert vid view vDefs, sDefs), a, f)

makeActive :: [Ident] -> Context -> Context
makeActive vids (Context (defs, _, fr) ss) =
  Context (defs, Set.fromList vids, fr) ss

addActive :: Ident -> Context -> Context
addActive vid (Context (defs, a, fr) ss) =
  Context (defs, Set.insert vid a, fr) ss

-- * Environment updaters

addViewDef :: Ident -> ViewType -> Salsa ()
addViewDef vid vDef =
  Salsa $ \(Context env shapes) ->
    let env' = insertViewDef vid vDef env
    in ((), Context env' shapes)

addShapeDef :: Ident -> Position -> Shape -> Salsa ()
addShapeDef sid pos sDef =
  Salsa $ \(Context env shapes) ->
    let env' = insertShapeDef sid sDef env
        (_,av,_) = env'
        shapePs = [(vid, pos) | vid <- Set.toList av]
        shapes' = insertShapePos sid shapePs shapes
    in ((), Context env' shapes')

clearActive :: Salsa()
clearActive = Salsa $ \c -> ((), makeActive [] c)

addView :: Ident -> Salsa ()
addView vid = Salsa $ \c -> ((), addActive vid c)

setActiveView :: Ident -> Salsa ()
setActiveView vid = do
  clearActive
  activate [vid]
  where
    activate :: [Ident] -> Salsa ()
    activate [] = return ()
    activate (v:vs) = do
      vDef <- liftC $ lookupViewDef v
      case vDef of
        Nothing ->
          error $ "Error: Undefined view " ++ show vid ++ " cannot be activated"
        Just (Vw _ _) -> do
          addView v
          activate vs
        Just (Gr gvids) ->
          activate $ gvids ++ vs


-- * Shape positions updater

setShapePos :: Ident -> ShapePos -> SalsaCommand ()
setShapePos sid ps = SalsaCommand $ \(Context env ss) ->
  ((), Context env $ insertShapePos sid ps ss)

--
-- Expressions
--

lowestCoords :: Ident -> SalsaCommand (Integer, Integer)
lowestCoords sid = do
  vps <- lookupShapePos sid
  case vps of
    Nothing ->
      error $ "Error: Unknow shape (projection failed): " ++ show sid
    Just ps ->
      case sort' $ unzip $ map snd ps of
        (x:_, y:_) -> return (x, y)
        _ ->
          error $ "Error: No position (projection failed): " ++ show sid
  where sort' (a,b) = (sort a, sort b)

expr :: Expr -> SalsaCommand Integer
expr (Plus e1 e2)   = liftM2 (+) (expr e1) (expr e2)
expr (Minus e1 e2)  = liftM2 (-) (expr e1) (expr e2)
expr (Const n)      = return n
expr (Xproj sid)    = liftM fst $ lowestCoords sid
expr (Yproj sid)    = liftM snd $ lowestCoords sid

--
-- Define the function command
--

command :: Command -> SalsaCommand ()
command (Move []  _) = return ()
command (Move (sid:sids) pos) = do
    activeViews <- getActive
    ps <- lookupShapePos sid
    ps' <- case ps of
      Nothing -> error $ "Error: Tried to move unknown shape: " ++ show sid
      Just xs -> do
        let q (vid, _) = vid `elem` activeViews
        let (xs1, other) = List.partition q xs
        xs2 <- mapM updatePos xs1
        return $ xs2 ++ other

    -- update context and do remaining sids
    setShapePos sid ps'
    command (Move sids pos)

    where
      updatePos :: (Ident, Position) -> SalsaCommand (Ident, Position)
      updatePos (vid, (x, y)) = case pos of
        (Abs e1 e2) -> do
          (x1, y1) <- doExpr e1 e2
          return (vid, (x1, y1))
        (Rel e1 e2) -> do
          (x1, y1) <- doExpr e1 e2
          return (vid, (x + x1, y + y1))
      doExpr e1 e2 = liftM2 (,) (expr e1) (expr e2)

command (At c vid) =
    local (makeActive [vid]) $ command c

command (Par c1 c2) = do
    -- this is "parallel" in as much as both commands gets to
    -- execute before next key frame is capturered.
    -- No check for illegal commands (e.g. overlapping manipulations)
    command c1
    command c2


--
-- Define the type Salsa
--

newtype Salsa a = Salsa { runS :: Context -> (a, Context) }
instance Monad Salsa where
  return x  = Salsa $ \c -> (x, c)
  m >>= f   = Salsa $ \c ->
                let (x, c') = runS m c
                in runS (f x) c'

--
-- Define the functions liftC, definition, and defCom
--

liftC :: SalsaCommand a -> Salsa a
```

```haskell
liftC (SalsaCommand sc) = Salsa $ \c@(Context env _) ->
  let (x, Context _ ss) = sc c
  in (x, Context env ss)

colour :: Colour -> String
colour Blue = "blue"
colour Plum = "plum"
colour Red = "red"
colour Green = "green"
colour Orange = "orange"

definition :: Definition -> Salsa ()
definition (Viewdef vid e1 e2) = do
  (w, h) <- liftC $ liftM2 (,) (expr e1) (expr e2)
  addViewDef vid $ Vw w h
  setActiveView vid

definition (Rectangle sid e1 e2 e3 e4 col) = do
  (x, y) <- liftC $ liftM2 (,) (expr e1) (expr e2)
  (w, h) <- liftC $ liftM2 (,) (expr e3) (expr e4)
  let c = colour col
  addShapeDef sid (x,y) $ Rect w h c

definition (Circle sid e1 e2 e3 col) = do
  (x, y) <- liftC $ liftM2 (,) (expr e1) (expr e2)
  r <- liftC $ expr e3
  let c = colour col
  addShapeDef sid (x,y) $ Cir r c

definition (View vid) = setActiveView vid
definition (Group gvid vids) = do
  addViewDef gvid $ Gr vids
  setActiveView gvid

defCom :: DefCom -> Salsa ()
defCom (Def def) = definition def
defCom (Com cmd) = liftC $ command cmd

-- Animate the n frames from key frame (ss0) to key frame (ss1);
-- includes both starting and end key frame. n > 0
animate :: Integer -> Shapes -> Shapes -> ShapeDefs -> [Frame]
animate n ss0 ss1 defs =
  map concat $ transpose [frames sid | sid <- sids]
  {-
    by list comprehension,
    we get 3 times n frames for drawing shapes x, y, z (in all views):
      [[xFrame_1, xFrame_2, ..., xFrame_n]
      ,[yFrame_1, yFrame_2, ..., yFrame_n]
      ,[zFrame_1, yFrame_2, ..., yFrame_n]]
    we need n frames:
      [xFrame_1 ++ yFrame_1 ++ zFrame_1
      ,xFrame_2 ++ yFrame_2 ++ zFrame_2
      ,...
      ,xFrame_n ++ yFrame_n ++ zFrame_n]
  -}
  where
    -- The shapes to be drawn, i.e. the ones defined in the end key frame
    sids :: [Ident]
    sids = Map.keys ss1

    -- Construct animation between ss0 and ss1 for sid, i.e. construct
    --    [xFrame_1, xFrame_2, ..., xFrame_n]
    -- for shape x. Each frame draws shape x in all applicable views
    frames :: Ident -> [Frame]
    frames sid =
      let
        -- we neeed instructions for drawing shape sid in each
        -- of these views from pos0 to pos1, for each view
        views =
          [(vid1, pos0, pos1)
           | (vid1, pos1) <- ps1
           , (vid0, pos0) <- ps0
           , vid0 == vid1]
        instrs = [drawInstrs shapeT p | p <- views]
      in
        case instrs of
          -- if shape is defined only in end key frame, we have no instrs
          -- until last frame
          [] -> replicate (fromInteger n - 1) [] ++
            [[appearInstr shapeT (vid1,pos1) | (vid1,pos1) <- ps1]]

          {-else we get 2 x n instr for drawing shape x in views A and B:
              instrs = [[xA_1, xA_2, ..., xA_end], [xB_1, xB_2, ..., xB_end]]
            we need n frames:
              frames = [[xA_1, xB_1], [xA_2, xB_2], ..., [xA_end, xB_end]] -}
          xs -> transpose xs

      where
        -- make n instrs for drawing shape sid in view vid from pos0 to pos1
        drawInstrs :: Shape -> (Ident, Position, Position) -> [GpxInstr]
        drawInstrs (Rect w h c) (vid, pos0, pos1) =
          [DrawRect x y w h vid c | (x, y) <- shapeSteps pos0 pos1]
        drawInstrs (Cir r c) (vid, pos0, pos1) =
          [DrawCirc x y r vid c | (x, y) <- shapeSteps pos0 pos1]

        -- make 1 instr for drawing shape sid in view vid at pos1
        appearInstr :: Shape -> (Ident, Position) -> GpxInstr
        appearInstr (Rect w h c) (vid,(x,y)) = DrawRect x y w h vid c
        appearInstr (Cir r c) (vid,(x,y)) = DrawCirc x y r vid c

        shapeSteps = interpolate n

        ps1 = fromMaybe (error "Error: Shape was not there!")
                        $ Map.lookup sid ss1
        ps0 = fromMaybe []  -- in this case, the shape was defined
                            -- after the command, and shows only
                            -- in the end key frame
                        $ Map.lookup sid ss0

        shapeT = fromMaybe
                   (error $ "Error: Shape definition missing: " ++ show sid)
                   $ Map.lookup sid defs

--
-- Define the function runProg
--

data KeyFrame = NoKeyFrame | KeyFrame Shapes

runProg :: Integer -> Program -> Animation
runProg n p
  | n < 1 =
```

```haskell
      error "Error: How should I animate anything with that frame rate?"
  | otherwise = fst $ runS (run p NoKeyFrame) (emptyContext n)

  where
    -- run program starting from key frame key0
    run :: Program -> KeyFrame -> Salsa Animation
    run [] NoKeyFrame = return ([], [])
    run [] (KeyFrame key0) = do
      key1 <- liftC getShapes -- capture final key frame
      sDefs <- liftC getShapeDefs
      let fs = animate n key0 key1 sDefs
      defs <- finalDefs
      return (defs, fs)

    run (Def def:dcs) kf = do
      defCom (Def def)
      (defs, fss) <- run dcs kf
      return (defs, fss)

    -- Commands indicate transitions between key frames
    run (Com cmd:dcs) kf = do
      sDefs <- liftC getShapeDefs
      curKey <- liftC getShapes
      let (n', key0, key1) =
            case kf of
              -- initial key frame is now fully defined; draw it
              NoKeyFrame -> (1, curKey, curKey)
              -- some prev key frame already drawn; draw from that
              KeyFrame k0 -> (n, k0, curKey)
      let fs = animate n' key0 key1 sDefs
      defCom (Com cmd)
      (defs, fss) <- run dcs (KeyFrame key1)
      return (defs, fs ++ fss)

    finalDefs :: Salsa [(ViewName, Integer, Integer)]
    finalDefs = do
      defs <- liftC getViewDefs
      return [(vid, w, h) | (vid, Vw w h) <- Map.toList defs]
```

Code piece 15: Tests in module `SalsaInterp`

```haskell
-- ============================================================================
--                                    TESTS
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
-- 1. QuickCheck for interpolate
-------------------------------------------------------------------------------


prop_pLength :: QC.NonNegative Integer -> Position -> Position -> Bool
prop_pLength (QC.NonNegative n) p1 p2 =
  length (interpolate n p1 p2) == fromInteger n

-- Only test non-empty lists
prop_pEndAtP2 :: QC.Positive Integer -> Position -> Position -> Bool
prop_pEndAtP2 (QC.Positive n) p1 p2 =
  last (interpolate n p1 p2) == p2

-- We must have that each list of coordinates move in same direction always
-- Tests only non-empty lists
prop_pOrdered :: QC.Positive Integer -> Position -> Position -> Bool
prop_pOrdered (QC.Positive n) p1 p2 =
  let ps = interpolate n p1 p2
      (xs, ys) = unzip ps
  in (isSorted xs || isSorted (reverse xs))
      && (isSorted ys || isSorted (reverse ys))
  where
    isSorted [] = True
    isSorted (z:zs) = fst $ foldl f (True, z) zs
    f (acc, x0) x1 = (acc && (x0 <= x1), x1)

-- Distances between points are about the same, i.e. ~= an n'th of the total
-- distance. Tests only non-empty lists
prop_pDistance :: QC.Positive Integer -> Position -> Position -> Bool
prop_pDistance (QC.Positive n) p_start p_end =
  let ps = p_start : interpolate n p_start p_end
      (dx', dy') = distance p_start p_end
      n' = fromInteger n
      d = (dx' / n', dy' / n') -- = the step in each coord
  in assertStepDistance d ps
  where
    -- Returns absolute distance split into dx and dy
    distance :: Position -> Position -> (Double, Double)
    distance (x1, y1) (x2, y2) =
      (fromInteger $ abs $ x2 - x1, fromInteger $ abs $ y2 - y1)

    assertStepDistance :: (Double, Double) -> [Position] -> Bool
    assertStepDistance _ [] = True
    assertStepDistance _ [_] = True
    assertStepDistance (dx, dy) (p1:p2:pss) =
      diffInBounds && assertStepDistance (dx, dy) (p2:pss)
      where
        (dx_i, dy_i) = distance p1 p2
        diffInBounds = (dx - 1.0 <= dx_i && dx_i <= dx + 1.0) &&
                       (dy - 1.0 <= dy_i && dy_i <= dy + 1.0)

runInterpolateTests :: IO ()
runInterpolateTests = do
  quickCheckWith stdArgs {maxSuccess = 10000} prop_pLength
  quickCheckWith stdArgs {maxSuccess = 10000} prop_pEndAtP2
  quickCheckWith stdArgs {maxSuccess = 10000} prop_pOrdered
  quickCheckWith stdArgs {maxSuccess = 10000} prop_pDistance


--------------------------------------------------------------------------------
-- 2. Unit tests for command
--------------------------------------------------------------------------------

tShape1orig :: (Ident, ShapePos)
tShape1orig = ("x", [("A", (50, 70))])
tShape2orig :: (Ident, ShapePos)
tShape2orig = ("y", [("A", (20, 200)), ("B", (65, 30))])

tCtxt :: Context
tCtxt =
 let
  -- Initial context for command unit test cases
  tViewDefs = Map.fromList [("A", Vw 300 300), ("B", Vw 600 400)]
  tShapeDefs = Map.fromList [("x", Cir 10 "blue"), ("y", Rect 25 42 "plum")]
  tActive = Set.fromList ["A"]
  tEnv = ((tViewDefs, tShapeDefs), tActive, 10)

  tShapes = Map.fromList [tShape1orig, tShape2orig]
 in Context tEnv tShapes

-- Command test cases
tCommands :: [Test]
tCommands =
 let
  (Context tEnv@((tViewDefs, tShapeDefs), _, _) _) = tCtxt

  -- Test on equality of contexts before and after single command
  cmdTest name inC outC testCmd =
    cmdTest' name inC outC (command testCmd)

  -- Test on equality of contexts before and after chain of commands
  cmdTest' name inC outC testCmd =
    TestCase $ assertEqual name (snd $ runSC testCmd inC) outC

  -- Test 1 - Absolute move on active view
  tShape1_move1 = ("x", [("A", (20, 20))])
  tCmd1 = cmdTest "Absolute move. Active view"
    tCtxt (Context tEnv $ Map.fromList [tShape1_move1, tShape2orig])
    $ Move ["x"] (Abs (Const 20) (Const 20))

  -- Test 2 - Relative move on active view
  tShape1_move2 = ("x", [("A", (70, 70))])
  tCmd2 = cmdTest "Relative move. Active view"
    tCtxt (Context tEnv $ Map.fromList [tShape1_move2, tShape2orig])
    $ Move ["x"] (Rel (Const 20) (Const 0))

  -- Test 3 - Move more than one
  tShape1_move3 = ("x", [("A", (100, 75))])
  tShape2_move1 = ("y", [("A", (70, 205)), ("B", (65, 30))])
  tCmd3 = cmdTest "Relative move of two shapes. Active view"
    tCtxt (Context tEnv $ Map.fromList [tShape1_move3, tShape2_move1])
    $ Move ["x", "y"] (Rel (Const 50) (Const 5))

  -- Test 3a - Move one shape in two active views
  tShape1_move3a = ("x", [("A", (50, 70))])
  tShape2_move1a = ("y", [("A", (70, 205)), ("B", (115, 35))])
  tCtxt3a@(Context tEnv3a _) = makeActive ["A", "B"] tCtxt
  tCmd3a = cmdTest "Relative move of one shape in two active views"
    tCtxt3a
    (Context tEnv3a $ Map.fromList [tShape1_move3a, tShape2_move1a])
    $ Move ["y"] (Rel (Const 50) (Const 5))

  -- Test 4 - Move in non-active view, followed by move in active
  tShape2_move2 = ("y", [("A", (42, 42)), ("B", (42, 42))])
  tCmd4_0 = do _ <- command $ At (Move ["y"] (Abs (Const 42) (Const 42))) "B"
               command $ Move ["y"] (Abs (Const 42) (Const 42))
  tCmd4 = cmdTest' "Relative move. Non-Active view"
    tCtxt (Context tEnv $ Map.fromList [tShape1orig, tShape2_move2])
    $tCmd4_0

  -- Test 5 - Chained '@'s; ignore all but first
  tShape2_move3 = ("y", [("B", (42, 42)), ("A", (20, 200))])
  tCmd5_0 = At (At (Move ["y"] (Abs (Const 42) (Const 42))) "B") "A"
  tActive5 = Set.fromList ["B"] -- This test tentatively changes active view
  tEnv5 = ((tViewDefs, tShapeDefs), tActive5, 10)
  tCmd5 = cmdTest
    "Abs move in non-active view; then chain of @s, no effect."
    tCtxt (Context tEnv5 $ Map.fromList [tShape1orig, tShape2_move3])
    $tCmd5_0

  -- Test 6 - Parallel commands
  tShape1_move4 = ("x", [("A", (30, 30))])
  tShape2_move4 = ("y", [("B", (42, 42)), ("A", (20, 200))])
  tCmd6_0 = Par (At (Move ["y"] (Abs (Const 42) (Const 42))) "B")
                (Move ["x"] (Abs (Const 30) (Const 30)))
  tCmd6 = cmdTest
    "Par cmd with move in non-active view and move in active view"
    tCtxt (Context tEnv $ Map.fromList [tShape1_move4, tShape2_move4])
    $tCmd6_0

  -- Advanced context for command unit test cases
  tViewDefs1 = Map.fromList [("A", Vw 300 300), ("B", Vw 600 400)
                            ,("C", Vw 500 200)]
  tShapeDefs1 = Map.fromList [("x", Cir 10 "blue")
                             ,("y", Rect 25 42 "plum")
                             ,("z", Cir 65 "orange")]
  tActive6 = Set.fromList ["A", "C"]
  tEnv6 = ((tViewDefs1, tShapeDefs1), tActive6, 10)
  tShape3_orig = ("z", [("B", (777, 888)), ("C", (1, 1))])
  tShapes1 = Map.fromList [tShape1orig, tShape2orig, tShape3_orig]
  tCtxt1 = Context tEnv6 tShapes1

  -- Test 7 - More parallel commands
  tShape1_move5 = ("x", [("A", (51, 71))])
  tShape3_move1 = ("z", [("C", (456, 789)), ("B", (999, 666))])
  tCmd7_0 = Par (Par (Move ["x"] (Rel (Const 1) (Const 1)))
                     (At (Move ["z"] (Abs (Const 999) (Const 666))) "B"))
                (Move ["z"] (Abs (Const 456) (Const 789)))
  tCmd7 = cmdTest
    "Par with nested par in active and non-active views"
    tCtxt1 (Context tEnv6 $ Map.fromList [tShape1_move5
                                         ,tShape2orig
                                         ,tShape3_move1])
    $tCmd7_0

 in
  [tCmd1
  ,tCmd2
  ,tCmd3
  ,tCmd3a
  ,tCmd4
  ,tCmd5
  ,tCmd6
  ,tCmd7
  ]
--------------------------------------------------------------------------------
-- 3. Unit tests for expr
--------------------------------------------------------------------------------

tExpressions :: [Test]
tExpressions =
 let
  -- Initial context for expression unit test cases (same as for commands)

  -- Test on expecting value of expression
  exprTest name inC outVal tExpr =
    TestCase $ assertEqual name (fst $ runSC (expr tExpr) inC) outVal

  -- Plus, minus and Const cases are trivial
  tExpr1 = exprTest "Constant" tCtxt 42 $ Const 42
  tExpr2 = exprTest "Plus constants" tCtxt 42
           $ Plus (Const 21) (Const 21)
  tExpr3 = exprTest "Plus constants" tCtxt 42
           $ Minus (Const 84) (Const 42)
  -- Projections
  tExpr4 = exprTest "Xproj with shape defined in one view"
            tCtxt 50 $ Xproj "x"
  tExpr5 = exprTest "Xproj with shape defined in more views"
            tCtxt 20 $ Xproj "y"
  tExpr6 = exprTest "Yproj with shape defined in one view"
            tCtxt 70 $ Yproj "x"
  tExpr7 = exprTest "Yproj with shape defined in more views"
            tCtxt 30 $ Yproj "y"
 in
  [tExpr1
  ,tExpr2
  ,tExpr3
  ,tExpr4
  ,tExpr5
  ,tExpr6
  ,tExpr7
  ]

--------------------------------------------------------------------------------
-- 4. Unit tests for definition
--------------------------------------------------------------------------------

-- Definition test cases
tDefinitions :: [Test]
tDefinitions =
 let
  -- Initial context for definition unit test cases
  tCtxt' = emptyContext 10
  (Context tEnv tShapes) = tCtxt'

  setAtv  a     ((v,s),_,fr)  = ((v,s),Set.fromList a,fr)
  setVDef v     ((_,s),a,fr)  = ((Map.fromList v,s),a,fr)
  addVDef (k,v) ((vs,s),a,fr) = ((Map.insert k v vs,s),a,fr)
  -- setSDef s     ((v,_),a,fr)  = ((v,Map.fromList s),a,fr)
```

```haskell
addSDef (k,s) ((v,ss),a,fr) = ((v,Map.insert k s ss),a,fr)

addSPos (k,ps)              = Map.insert k ps

-- Test on equality of contexts; single definition
defTest name inC outC testDef =
  defTest' name inC outC (definition testDef)

-- Test on equality of contexts; more definitions
defTest' name inC outC testDef =
  TestCase $ assertEqual name outC $ snd $ runS testDef inC

-- Test 1 - View definition in empty environment
tEnv1 = setAtv ["A"] $ setVDef [("A", Vw 200 100)] tEnv
tDef1 = defTest "Viewdef in empty context"
  tCtxt' (Context tEnv1 tShapes)
  $ Viewdef "A" (Const 200) (Const 100)

-- Test 2 - View definition in non-empty environment
tEnv2 = setAtv ["B"] $
        setVDef [("B", Vw 500 300), ("A", Vw 200 100)] tEnv
tDef2 = defTest "Viewdef in non-empty context"
  (Context tEnv1 tShapes) (Context tEnv2 tShapes)
  $ Viewdef "B" (Const 500) (Const 300)

-- Test 3 - Making single view active when already active
tDef3 = defTest "Single view active when already active"
  (Context tEnv2 tShapes) (Context tEnv2 tShapes)
  $ View "B"

-- Test 4 - Making single view active when non-active
tEnv4 = setAtv ["A"] tEnv2
tDef4 = defTest "Single view active when non-active"
  (Context tEnv2 tShapes) (Context tEnv4 tShapes)
  $ View "A"

-- Test 5 - Defining group of all previously defined views
tEnv5 = setAtv ["A", "B"] $
        addVDef ("X", Gr ["A", "B"]) tEnv4
tDef5 = defTest "Group of two views"
  (Context tEnv4 tShapes) (Context tEnv5 tShapes)
  $ Group "X" ["A", "B"]

-- Test 6 - Defining group of one previously defined views
tEnv6 = setAtv ["B"] $
        addVDef ("Y", Gr ["B"]) tEnv5
tDef6 = defTest "Group of one"
  (Context tEnv5 tShapes) (Context tEnv6 tShapes)
  $ Group "Y" ["B"]

-- Test 7 - Chaining defitions, context updates
tEnv7 = setAtv ["D"] $
        addVDef ("C", Vw 1000 4200) $
        addVDef ("D", Vw 0 0) tEnv6
tDef7 = defTest' "First viewdef, then another"
  (Context tEnv6 tShapes) (Context tEnv7 tShapes)
  (do definition $ Viewdef "C" (Const 1000) (Const 4200)
      definition $ Viewdef "D" (Const 0) (Const 0))

-- Test 8 - Defining group consisting of view and other group
tEnv8 = setAtv ["A", "B", "C"] $
```

```
        addVDef ("Z", Gr ["X", "C"]) tEnv7
tDef8 = defTest "Group of one view, one other group"
  (Context tEnv7 tShapes) (Context tEnv8 tShapes)
  $ Group "Z" ["X", "C"]

-- Test 9 - Defining group with duplicates
tEnv9 = setAtv ["A", "B"] $
        addVDef ("W", Gr ["X", "Y"]) tEnv8
tDef9 = defTest "Group of two groups, with duplicates"
  (Context tEnv8 tShapes) (Context tEnv9 tShapes)
  $ Group "W" ["X", "Y"]

-- Test 10 - Groups of groups of groups
tEnv10 = setAtv ["A", "C", "B", "D"] $
         addVDef ("V", Gr ["D", "Z"]) tEnv9
tDef10 = defTest "Group of groups of groups"
  (Context tEnv9 tShapes) (Context tEnv10 tShapes)
  $ Group "V" ["D", "Z"]

-- Test 11 - Rectangle in one active view
tEnv11 = addSDef ("s", Rect 100 200 "blue") tEnv7
tShapes11 = addSPos ("s", [("D", (1,2))]) tShapes
tDef11 = defTest "Rectangle, one active view"
  (Context tEnv7 tShapes) (Context tEnv11 tShapes11)
  $ Rectangle "s" (Const 1) (Const 2) (Const 100) (Const 200) Blue

-- Test 12 - Circle in one active view
tEnv12 = addSDef ("t", Cir 42 "plum") tEnv11
tShapes12 = addSPos ("t", [("D", (3,4))]) tShapes11
tDef12 = defTest "Circle, one active view"
  (Context tEnv11 tShapes11) (Context tEnv12 tShapes12)
  $ Circle "t" (Const 3) (Const 4) (Const 42) Plum

-- Test 13 - Rectangle in two active views
tEnv13 = addSDef ("s", Rect 100 200 "blue") tEnv9
tShapes13 = addSPos ("s", [("A", (1,2))
                          ,("B", (1,2))]) tShapes
tDef13 = defTest "Rectangle, two active views"
  (Context tEnv9 tShapes) (Context tEnv13 tShapes13)
  $ Rectangle "s" (Const 1) (Const 2) (Const 100) (Const 200) Blue

 in
  [tDef1
  ,tDef2
  ,tDef3
  ,tDef4
  ,tDef5
  ,tDef6
  ,tDef7
  ,tDef8
  ,tDef9
  ,tDef10
  ,tDef11
  ,tDef12
  ,tDef13
  ]

--------------------------------------------------------------------------------
-- 5. Unit tests runs
--------------------------------------------------------------------------------
tests :: Test
tests = TestList
  [TestLabel "Commands"    $ TestList tCommands
  ,TestLabel "Expressions" $ TestList tExpressions
  ,TestLabel "Definitions" $ TestList tDefinitions
  ]

runHUTests :: IO Counts
runHUTests = runTestTT tests


--------------------------------------------------------------------------------
-- 6. QuickCheck for animate
--------------------------------------------------------------------------------

-- tProg :: Program
-- tProg = [ Def (Viewdef "One" (Const 500) (Const 500))
        -- , Def (Viewdef "Two" (Const 400) (Const 400))
        -- , Def (Group "Both" ["One","Two"])
        -- , Def (View "Both")
        -- , Def (Rectangle "larry" (Const 10) (Const 350)
                                 -- (Const 20) (Const 20) Blue)
        -- , Def (Rectangle "fawn" (Const 300) (Const 350)
                                -- (Const 15) (Const 25) Plum)
        -- , Def (View "Two")
        -- , Com (Par (Move ["larry"] (Abs (Const 300) (Const 350)))
                    -- (Move ["fawn"] (Abs (Const 10) (Const 350))))
        -- , Def (View "Both")
        -- , Com (Move ["larry","fawn"]
                     -- (Rel (Const 0) (Minus (Const 0) (Const 300))))]

-- We test only for limited number of views, shapes, since the algorithm
-- should be correct for larger numbers as well, if it is for smaller

-- Max length of generated identifiers (we remove duplicates before use)
identSize :: Int
identSize = 5

videntChars :: String
videntChars = ['A'..'Z']
sidentChars :: String
sidentChars = ['a'..'z']

getIdent :: String -> Gen Ident
getIdent chars = vectorOf identSize $ QC.elements chars

getIdents :: String -> Gen [Ident]
getIdents chars = do
  n <- choose (1, 100)
  vids <- vectorOf n $ getIdent chars
  return $ List.nub vids -- remove dublicates, but still at least 1

newtype TestPos = TestPos Position
instance Arbitrary TestPos where
  arbitrary = liftM TestPos $ liftM2 (,) getn getn
    where getn = do { (QC.NonNegative n) <- arbitrary; return n }

getShape :: [Ident] -> Gen ((Ident, ShapePos)
                           ,(Ident, ShapePos)
                           ,(Ident, Shape))
getShape vids = do
    sid <- getIdent sidentChars
```

```haskell
    def'ed_on <- subsequence vids
    ps0 <- vectorOf (length def'ed_on) arbitrary
    ps1 <- vectorOf (length def'ed_on) arbitrary
    let pss0 = zip def'ed_on [p | (TestPos p) <- ps0]
    let pss1 = zip def'ed_on [p | (TestPos p) <- ps1]

    shape <- oneof [aRect, aCir]

    return ((sid, pss0), (sid, pss1), (sid, shape))

    where
      -- can be invalid shapedefs, but serves our purpose
      aRect = liftM3 Rect arbitrary arbitrary colors
      aCir  = liftM2 Cir arbitrary colors
      colors = elements ["blue", "plum", "red", "green", "orange"]

newtype SubSeq = SubSeq ([Ident] -> Gen [Ident])
instance Arbitrary SubSeq where
  arbitrary = return $ SubSeq subsequence

distro :: [Integer]
distro = [0,0,1] -- 2-1 for not picking

subsequence :: [Ident] -> Gen [Ident]
subsequence [] = return [] -- Bad Thing to have happen!
subsequence (x:xs) = do
  selectBits <- vectorOf (length xs) $ elements distro
  return $ x: [x' | (1, x') <- zip selectBits xs] -- ensure at least one

newtype TestKeyFrames = TestKeyFrames (Shapes, Shapes, ShapeDefs)
  deriving (Eq, Show)
instance Arbitrary TestKeyFrames where
  arbitrary = do
    vids <- getIdents videntChars

    n <- choose (1, 200)
    ss <- vectorOf n $ getShape vids
    let (ss0, ss1, sdefs) = unzip3 ss

    return $ TestKeyFrames (Map.fromList ss0
                           ,Map.fromList ss1
                           ,Map.fromList sdefs)

prop_pAnimationLength :: QC.Positive Integer -> TestKeyFrames -> Bool
prop_pAnimationLength (QC.Positive n) (TestKeyFrames (ss0, ss1, defs)) =
  length (animate n ss0 ss1 defs) == fromInteger n

-- For this property, we only care about positive n
prop_pFrameLength :: QC.Positive Integer -> TestKeyFrames -> Bool
prop_pFrameLength (QC.Positive n) (TestKeyFrames (ss0, ss1, defs)) =
  let rs = animate n ss0 ss1 defs
      m = length $ concatMap snd (Map.toList ss1)
      p fr = length fr == m
  in all p rs

-- For this property, we only care about positive n
prop_pInstructions :: QC.Positive Integer -> TestKeyFrames -> Bool
prop_pInstructions (QC.Positive n) (TestKeyFrames (ss0, ss1, defs)) =
  let ss = Map.toList ss1
  in all q ss   -- forall s in S
  where
    rs = animate n ss0 ss1 defs

    -- filter frame for specific shape
    f :: Shape -> GpxInstr -> Bool
    f (Rect w h c) (DrawRect _ _ w' h' _ c') = w' == w && h' == h && c' == c
    f (Rect {}) _ = False
    f (Cir r c) (DrawCirc _ _ r' _ c') = r' == r && c' == c
    f (Cir {} ) _ = False

    p :: Shape -> Int -> Frame -> Bool
    p s m fr = m <= length (filter (f s) fr)

    q :: (Ident, ShapePos) -> Bool
    q (s, ps) = all (p s1 $ length ps) rs
      where
        s1 = fromMaybe (error "Test failed") (Map.lookup s defs)

runAnimateTests :: IO ()
runAnimateTests = do
  quickCheckWith stdArgs {maxSuccess = 1000} prop_pAnimationLength
  quickCheckWith stdArgs {maxSuccess = 1000} prop_pFrameLength
  quickCheckWith stdArgs {maxSuccess = 500} prop_pInstructions

--------------------------------------------------------------------------------
-- 7. Run all tests
--------------------------------------------------------------------------------

runTests :: IO Counts
runTests = do
  runInterpolateTests
  runAnimateTests
  runHUTests
```

Code piece 16: Code for running some small Salsa programs

```haskell
{-
  Various test programs to compile and interpret
-}
module SalsaTest where

import qualified SalsaParser as SP
import qualified SalsaInterp as SI
import Gpx

tProgram1 :: String
tProgram1 =
  "viewdef Default 400 400 \
  \rectangle box 10 400 20 20 green \
  \box -> (10, 200) \
  \box -> +(100, 0) \
  \box -> (110,400) \
  \box -> +(0-100, 0)"

tProgram2 :: String
tProgram2 =
  "viewdef One 500 500 \
  \viewdef Two 400 400 \
  \group Both [One Two] \
  \view Both \
  \rectangle larry 10 350 20 20 blue \
  \rectangle fawn 300 350 15 25 plum \
  \view Two \
  \larry -> (300, 350) || fawn -> (10,350) \
  \view Both \
  \larry fawn -> +(0, 0 - 300)"

tProgram3 :: String
tProgram3 =
  "viewdef Main 400 400 \
  \circle a 10 200 50 red \
  \a -> (200, 350) \
  \rectangle b 20 20 100 15 blue \
  \rectangle c 380 400 10 300 plum \
  \c -> (200, 350) || a -> +(0-100, 0) \
  \rectangle d 195 205 10 10 green "

runWrite :: FilePath -> String -> Integer -> IO ()
runWrite filename p n = writeFile filename $ show $ run p n

run :: String -> Integer -> Animation
run p n =
  case SP.parseString p of
    Right ast -> SI.runProg n ast
    Left _ -> error "Parsing failed"
```

Code piece 17: Type definitions for abstract syntax tree of Salsa programs

```
module SalsaAst where

type Program = [DefCom]
data DefCom = Def Definition
            | Com Command
            deriving (Show, Eq)
data Definition = Viewdef Ident Expr Expr
                | Rectangle Ident Expr Expr Expr Expr Colour
                | Circle Ident Expr Expr Expr Colour
                | View Ident
                | Group Ident [Ident]
                deriving (Show, Eq)
data Command = Move [Ident] Pos
             | At Command Ident
             | Par Command Command
             deriving (Show, Eq)
data Pos = Abs Expr Expr
         | Rel Expr Expr
         deriving (Show, Eq)
data Expr = Plus Expr Expr
          | Minus Expr Expr
          | Const Integer
          | Xproj Ident
          | Yproj Ident
          deriving (Show, Eq)
data Colour = Blue | Plum | Red | Green | Orange
            deriving (Show, Eq)
type Ident = String
```

Code piece 18: Type definitions for low-level animations

```
module Gpx where

type ViewName = String
type ColourName = String
type Frame = [GpxInstr]
type Animation = ([(ViewName, Integer, Integer)], [Frame])
data GpxInstr = DrawRect Integer Integer Integer Integer ViewName ColourName
              | DrawCirc Integer Integer Integer ViewName ColourName
              deriving (Eq, Show)
```

Code piece 19: Atomic Transaction Server (at_server)

```
%%%-------------------------------------------------------------------
%%% @author Michael Kirkedal Thomsen <shapper@diku.dk>
%%% @copyright (C) 2013, Michael Kirkedal Thomsen
%%% @doc
%%% Hand-in for AP Exam 2013.
%%% Implementation of the atomic transaction server
%%% @end
%%% Created : Oct 2013 by Michael Kirkedal Thomsen <shapper@diku.dk>
%%%-------------------------------------------------------------------
%%% Student name: Jonas Stig Kaempf Hansen
%%% Student KU-id:
%%%-------------------------------------------------------------------

-module(at_server).
-include_lib("eunit/include/eunit.hrl").

-export([start/1, stop/1, begin_t/1, doquery/2, query_t/3, update_t/3, commit_t/2]).


% debug(Msg, Args) ->
  % io:format("[DEBUG - ~p]: " ++ Msg, [self()|Args]).
debug(_, _) -> ok.

%%%-------------------------------------------------------------------
%%% API
%%%-------------------------------------------------------------------

start(State) ->
    {ok, spawn(fun() -> initATServer(State) end)}.

% Blocking
stop(AT) ->
  call_stop(AT).

% Blocking
doquery(AT, Fun) ->
  call_query(AT, Fun).

% Non-blocking
begin_t(AT) ->
  Ref = make_ref(),
  cast_begin_t(AT, Ref),
  {ok, Ref}.

% Blocking
query_t(AT, Ref, Fun) ->
  call_query_t(AT, Ref, Fun).

% Non-blocking
update_t(AT, Ref, Fun) ->
  cast_update_t(AT, Ref, Fun).

% Blocking
commit_t(AT, Ref) ->
  call_commit_t(AT, Ref).

%%%-------------------------------------------------------------------
%%% Communication primitives
%%%-------------------------------------------------------------------
```

```
%% synchronous communication

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} -> Response
    end.

reply(From, Msg) ->
    From ! {self(), Msg}.

reply_ok(From) ->
    reply(From, ok).

reply_ok(From, Msg) ->
    reply(From, {ok, Msg}).

reply_error(From) ->
    reply(From, error).

reply_abort(From) ->
    reply(From, aborted).

call_stop(AT) -> rpc(AT, stop).
call_query(To, Fun) -> rpc(To, {doquery, Fun}).
call_query_t(AT, Ref, Fun) -> rpc(AT, {query_t, Ref, Fun}).
call_commit_t(AT, Ref) -> rpc(AT, {commit_t, Ref}).

%% asynchronous communication

info(Pid, Msg) ->
    Pid ! Msg.

cast_begin_t(AT, Ref) ->
  info(AT, {begin_t, Ref}).
cast_update_t(AT, Ref, Fun) ->
  info(AT, {update_t, Ref, Fun}).
cast_update(TID, Ref, Fun) ->
  info(TID, {self(), {update, Ref, Fun}}).
cast_update_error(AT, Ref) ->
  info(AT, {updateFailed, Ref}).
cast_stop(TID) ->
  info(TID, {self(), stop}).

%%%-------------------------------------------------------------------
%%% Internal Implementation
%%%-------------------------------------------------------------------

%%% AT Server loop

initATServer(State) ->
  atServerLoop(State, []).

atServerLoop(State, Trans) ->
  receive
    {From, {doquery, Fun}} ->
      Result = safeQuery(Fun, State),
      reply(From, Result),
      atServerLoop(State, Trans);

    {begin_t, Ref} ->
```

```
      Trans1 = [{Ref, newTrans(State)} | Trans],
      atServerLoop(State, Trans1);

    {From, {query_t, Ref, Fun}} ->
      TID = lookupTransaction(Ref, Trans),
      Trans1 =
        case queryTransaction(TID, Fun) of
          error ->
            reply_abort(From),
            abortTransaction(Ref, Trans);

          {ok, Result} ->
            reply_ok(From, Result),
            Trans
        end,
      atServerLoop(State, Trans1);

    {update_t, Ref, Fun} ->
      TID = lookupTransaction(Ref, Trans),
      updateTransaction(TID, Ref, Fun),
      atServerLoop(State, Trans);

    {updateFailed, Ref} ->
      Trans1 = abortTransaction(Ref, Trans),
      atServerLoop(State, Trans1);

    {From, {commit_t, Ref}} ->
      TID = lookupTransaction(Ref, Trans),
      Id = fun (X) -> X end,
      {State1, Trans1} =
        case queryTransaction(TID, Id) of
          error ->
            reply_abort(From),
            {State, abortTransaction(Ref, Trans)};
          {ok, S} ->
            reply_ok(From),
            {S, abortAllTransactions(Trans)}
        end,
      atServerLoop(State1, Trans1);

    {From, stop} ->
      abortAllTransactions(Trans),
      reply_ok(From, State)
  end.

%%% Transaction loop

initTransaction(Server, State) ->
  transactionLoop(Server, State).

transactionLoop(Server, State) ->
  receive
    {Server, {doquery, Fun}} ->
      debug("Query received~n", []),
      reply(Server, safeQuery(Fun, State)),
      transactionLoop(Server, State);

    {Server, {update, Ref, Fun}} ->
      case safeUpdate(Fun, State) of
        error ->
          cast_update_error(Server, Ref),
          transactionFailed(Server);
        {ok, State1} ->
          transactionLoop(Server, State1)
      end;

    {Server, stop} ->
      debug("Transaction stopping in State ~p~n", [State]),
      ok
  end.

transactionFailed(Server) ->
  receive
    {Server, {doquery, _}} ->
      reply_error(Server),
      transactionFailed(Server);
    {Server, stop} -> ok
  end.

%%% Helper functions

% Returns {ok, Fun(State)}, or error if Fun(State) fails
safeQuery(Fun, State) ->
  try Fun(State) of
    Result ->
      debug("Query on State ~p successful. Result: ~p.~n", [State, Result]),
      {ok, Result}
  catch
    _ : Why ->
      debug("Query on State ~p failed. Reason: ~p~n", [State, Why]),
      error
  end.

% Returns the updated state as {ok, State1}, or error
safeUpdate(Fun, State) -> safeQuery(Fun, State).

% spawn new transaction
newTrans(State) ->
  Server = self(),
  spawn(fun() -> initTransaction(Server, State) end).

% Lookup transaction Ref and get transaction process id
lookupTransaction(_, []) -> aborted;
lookupTransaction(Ref, [{Ref, TID} | _]) -> TID;
lookupTransaction(Ref, [_ | Trans]) -> lookupTransaction(Ref, Trans).

% Abort transaction Ref from list of transactions
abortTransaction(_, []) -> [];
abortTransaction(Ref, [{Ref, TID} | Trans]) ->
  cast_stop(TID),
  Trans;
abortTransaction(Ref, [T | Trans]) ->
  [T | abortTransaction(Ref, Trans)].

abortAllTransactions([]) -> [];
abortAllTransactions([{_, TID} | Trans]) ->
  cast_stop(TID),
  abortAllTransactions(Trans).

% Queries a transaction, get either {ok, Result}, or error
queryTransaction(aborted, _) -> error;
queryTransaction(TID, Fun) -> call_query(TID, Fun).
```

```
updateTransaction(aborted, _, _) -> error;
updateTransaction(TID, Ref, Fun) ->
  cast_update(TID, Ref, Fun).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% BLACK BOX TESTING OF API
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Returns tuple of functions \x -> A,
% where A is (in the order returned) {x, x + 1, x + bad, throw(fit)}
% Suggested use: {Id, Inc, Fail, Throw} = basic_funs()
basic_funs() ->
  {fun(X) -> X end
  ,fun(X) -> X + 1 end
  ,fun(X) -> X + bad end
  ,fun(_) -> throw(fit) end}.

%%% Test cases

start_server_test() ->
  {ok, S} = start(0), stop(S).

simple_stop_server_test() ->
  {ok, Server} = start(42),
  {ok, 42} = stop(Server).

do_single_query_test() ->
  {ok, Server} = start(0),
  {_, Inc, _, _} = basic_funs(),
  {ok, 1} = doquery(Server, Inc),
  {ok, 0} = stop(Server).

do_more_query_test() ->
  {ok, Server} = start(0),
  {_, Inc, _, _} = basic_funs(),
  {ok, 1} = doquery(Server, Inc),
  {ok, 1} = doquery(Server, Inc),
  {ok, 0} = stop(Server).

fail_fun_doquery_test() ->
  {ok, Server} = start(0),
  {_, _, Wrong, _} = basic_funs(),
  error = doquery(Server, Wrong),
  {ok, 0} = stop(Server).

throw_fun_doquery_test() ->
  {ok, Server} = start(0),
  {_, _, _, Throw} = basic_funs(),
  error = doquery(Server, Throw),
  {ok, 0} = stop(Server).

% Note this is not intended as testing of general uniqueness of refs
begin_t_test() ->
  {ok, Server} = start(42),
  {ok, Ref1} = begin_t(Server),
  {ok, Ref2} = begin_t(Server),
  ?assert(Ref1 =/= Ref2),
  {ok, 42} = stop(Server).
```

```
query_t_test() ->
  {ok, Server} = start(42),

  % Transaction 1
  {ok, Ref1} = begin_t(Server),
  {_, Inc, Wrong, Fail} = basic_funs(),
  {ok, 43} = query_t(Server, Ref1, Inc),

  aborted = query_t(Server, Ref1, Wrong),
  aborted = query_t(Server, Ref1, Inc),

  % Transaction 2
  {ok, Ref2} = begin_t(Server),
  {ok, 43} = query_t(Server, Ref2, Inc),

  aborted = query_t(Server, Ref2, Fail),
  aborted = query_t(Server, Ref2, Inc),

  {ok, 42} = stop(Server).

update_t_test() ->
  {ok, Server} = start(42),
  {Id, Inc, Fail, Throw} = basic_funs(),

  {ok, Ref1} = begin_t(Server),

  update_t(Server, Ref1, Id),
  {ok, 42} = query_t(Server, Ref1, Id),

  update_t(Server, Ref1, Inc),
  {ok, 43} = query_t(Server, Ref1, Id),
  {ok, 42} = doquery(Server, Id),

  update_t(Server, Ref1, Fail),
  aborted = query_t(Server, Ref1, Id),
  {ok, 42} = doquery(Server, Id),

  {ok, Ref2} = begin_t(Server),
  {ok, 42} = query_t(Server, Ref2, Id),
  update_t(Server, Ref2, Throw),
  aborted = query_t(Server, Ref2, Id),
  {ok, 42} = doquery(Server, Id),

  % update on aborted, does not crash anything
  update_t(Server, Ref2, Inc),
  {ok, 42} = doquery(Server, Id),

  {ok, 42} = stop(Server).

commit_t_test() ->
  {ok, Server} = start(42),
  {Id, Inc, Fail, _} = basic_funs(),

  {ok, Ref1} = begin_t(Server),
  {ok, Ref2} = begin_t(Server),

  update_t(Server, Ref1, Inc),
  update_t(Server, Ref1, Inc), % Ref1 == 44
  update_t(Server, Ref2, Inc), % Ref2 == 43

  {ok, 42} = doquery(Server, Id),
  ok = commit_t(Server, Ref1),
  aborted = commit_t(Server, Ref2),
  {ok, 44} = doquery(Server, Id),
  aborted = commit_t(Server, Ref1),

  aborted = query_t(Server, Ref1, Id),
  aborted = query_t(Server, Ref2, Id),

  {ok, Ref3} = begin_t(Server),
  update_t(Server, Ref3, Fail),
  aborted = commit_t(Server, Ref3),

  {ok, 44} = stop(Server).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% PARTIAL WHITE BOX TESTING OF INTERNAL IMPLEMENTATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lookup_t_test() ->
  aborted = lookupTransaction(42, []),
  Ts = [{4, hitchhikers}, {2, guide}, {6, galaxy}],
  hitchhikers = lookupTransaction(4, Ts),
  guide = lookupTransaction(2, Ts),
  galaxy = lookupTransaction(6, Ts),
  aborted = lookupTransaction(7, Ts).

abort_t_test() ->
  [] = abortTransaction(42, []),
  TID = self(),
  Ts = [{4, TID}, {2, TID}, {100, TID}],

  Ts = abortTransaction(0, Ts), % abort aborted trans

  Ts1 = abortTransaction(4, Ts), % abort head of list
  [{2, TID}, {100, TID}] = Ts1,

  Ts1 = abortTransaction(4, Ts1), % abort same again

  Ts2 = abortTransaction(2, Ts), % abort some middle
  [{4, TID}, {100, TID}] = Ts2,

  Ts3 = abortTransaction(100, Ts), % abort tail
  [{4, TID}, {2, TID}] = Ts3,

  Ts4 = abortTransaction(2, Ts3), % abort last of two
  [{4, TID}] = Ts4,

  Ts5 = abortTransaction(4, Ts4), % single
  [] = Ts5.
```

Code piece 20: Extensions for Atomic Server API (at_extapi)

```
%%%-------------------------------------------------------------------
%%% @author Michael Kirkedal Thomsen <shapper@diku.dk>
%%% @copyright (C) 2013, Michael Kirkedal Thomsen
%%% @doc
%%% Hand-in for AP Exam 2013.
%%% Implementation of the atomic transaction server
%%% @end
%%% Created : Oct 2013 by Michael Kirkedal Thomsen <shapper@diku.dk>
%%%-------------------------------------------------------------------
%%% Student name: Jonas Stig Kaempf Hansen
%%% Student KU-id:
%%%-------------------------------------------------------------------

-module(at_extapi).
-include_lib("eunit/include/eunit.hrl").

-export([abort/2, tryUpdate/2, ensureUpdate/2, choiceUpdate/3, choiceUpdate/4]).

-define(SERVER_MODULE, at_server).
-import(?SERVER_MODULE, [start/1, stop/1, begin_t/1, doquery/2,
                         query_t/3, update_t/3, commit_t/2]).

%%%-------------------------------------------------------------------
%%% Extended API
%%%-------------------------------------------------------------------

abort(AT, Ref) ->
  Fail = fun (_) -> throw(abortMe) end,
  update_t(AT, Ref, Fail).

tryUpdate(AT, Fun) ->
  Me = self(),
  {ok, Ref} = begin_t(AT),
  Upd = fun (S) ->
    try Fun(S) of
      S1 ->
        Me ! {updated, Ref},
        S1
    catch
      _ : Why ->
        Me ! {failed, Ref},
        throw(Why)
    end
  end,
  update_t(AT, Ref, Upd),
  receive
    {updated, Ref} -> commit_t(AT, Ref);
    {failed, Ref} -> error
  end.

ensureUpdate(AT, Fun) ->
  case tryUpdate(AT, Fun) of
    aborted ->
      ensureUpdate(AT, Fun);
    X -> X % is either error or ok, both of which
           % are also acceptable results here
  end.

choiceUpdate(AT, Fun, Val_list) ->
  choiceUpdateWork(AT, Fun, Val_list),
  receive % first one that messages back, wins
    {committed_t, Result} -> Result
  end.

choiceUpdate(AT, Fun, Val_list, Timeout) ->
  choiceUpdateWork(AT, Fun, Val_list),
  receive % first one that messages back, wins
    {committed_t, Result} -> Result
  after
    Timeout -> timeout
  end.

% Long, and complicated version
% commitWorker(AT, Ref, Parent) ->
  % CommitRes = commit_t(AT, Ref),
  % Parent ! {committed_t, CommitRes}.
% choiceUpdateWork(AT, Fun, Val_list) ->
  % TransactionWorker = % produce a fun to apply to state within transaction process
    % fun (Parent, E, Ref) ->
      % fun(S) -> % this works inside the transaction process
        % Res = Fun(S, E), % may fail, but let transaction handle it
        % % spawn someone to commit (we cannot do it, for risk of deadlock!),
        % spawn(fun()-> commitWorker(AT, Ref, Parent) end),
        % % and then return my new state
        % Res
        % % even though we spawn now, server cannot complete the commit
        % % before we finish, so state will be updated for commit
      % end
    % end,

  % Me = self(),
  % Launch =
    % fun(Parent, E) ->
      % {ok, Ref} = begin_t(AT),
      % update_t(AT, Ref, TransactionWorker(Parent, E, Ref))
    % end,
  % _ = [spawn(fun() -> Launch(Me, E) end) || E <- Val_list].

% Simpler version, works
choiceUpdateWork(AT, Fun, Val_list) ->
  Launch = fun(Parent, E) ->
    Fun1 = fun(S) -> Fun(S, E) end,
    case tryUpdate(AT, Fun1) of
      error -> ok; % update failed, silently die
      CommitRes -> Parent ! {committed_t, CommitRes}
    end
  end,
  Me = self(),
  _ = [spawn(fun() -> Launch(Me, E) end) || E <- Val_list].

%%%-------------------------------------------------------------------
%%% BLACK-BOX UNIT TESTS FOR EXTAPI
%%%-------------------------------------------------------------------

basic_funs() ->
  {fun(X) -> X end
  ,fun(X) -> X + 1 end
  ,fun(X) -> X + bad end
  ,fun(_) -> throw(fit) end}.

abort_test() ->
```

```
  {Id, _, _, _} = basic_funs(),
  {ok, Server} = start(42),
  {ok, Ref} = begin_t(Server),

  abort(Server, Ref),
  aborted = query_t(Server, Ref, Id),

  {ok, 42} = stop(Server).

tryUpdate_test() ->
  {_, Inc, Fail, Throw} = basic_funs(),
  {ok, Server} = start(42),
  error = tryUpdate(Server, Fail),
  error = tryUpdate(Server, Throw),
  ok = tryUpdate(Server, Inc),

  Me = self(),
  Master = spawn(fun()->
    receive {hello, Slave} ->
      tryUpdate(Server, Inc),
      Slave ! goAhead
    end
  end),

  SlowUpd = fun(X) ->
    Master ! {hello, self()},
    receive goAhead -> X + 1 end

  end,
  spawn(fun() -> Me ! tryUpdate(Server, SlowUpd) end),

  receive
    X -> ?assert(X =:= aborted)
  end,

  {ok, 44} = stop(Server).

ensureUpdate_test() ->
  {_, Inc, Fail, Throw} = basic_funs(),
  {ok, Server} = start(42),
  error = ensureUpdate(Server, Fail),
  error = ensureUpdate(Server, Throw),
  ok = ensureUpdate(Server, Inc), % server state = 43

  Me = self(),
  Master = spawn(fun()->
    receive
      Slave ->
        ensureUpdate(Server, Inc),
        Slave ! goAhead,
        receive NewSlave -> NewSlave ! goAhead end
    end
  end),

  SlowUpd = fun(X) -> Master ! self(), receive goAhead -> X + 1 end end,
  spawn(fun() -> Me ! ensureUpdate(Server, SlowUpd) end),

  receive
    X -> ?assert(X =:= ok)
  end,
  {ok, 45} = stop(Server).

% To ensure deterministic testing, we must have some updates wait for others
% Testing assumes that at least one function successfully updates the state
choiceUpdate_test() ->
  {Id, _, _, _} = basic_funs(),
  {ok, Server} = start(42),

  Me = self(),
  Master = spawn(fun() -> receive {two, PID} -> PID ! goAhead end end),
  Choices = [
    fun(S) -> Me     ! {one, self()}, receive goAhead -> S + 1 end end,
    fun(S) -> Master ! {two, self()}, receive goAhead -> S + 2 end end,
    fun(S) -> Me     ! {three, self()}, receive goAhead -> S + 3 end end],
  Fun = fun(S, E) -> E(S) end,

  ok = choiceUpdate(Server, Fun, Choices),
  {ok, 44} = doquery(Server, Id),

  % % make remaining finish
  receive {one, PID1} -> PID1 ! goAhead end,
  receive {three, PID3} -> PID3 ! goAhead end,
  {ok, 44} = stop(Server).

choiceUpdate2_test() ->
  {ok, Server} = start(42),

  Me = self(),
  Master = spawn(fun() ->
    receive ready -> receive {two, PID} -> PID ! goAhead end end end),
  Choices = [
    fun(S) -> Me     ! {one, self()}, receive goAhead -> S + 1 end end,
    fun(S) -> Master ! {two, self()}, receive goAhead -> S + 2 end end,
    fun(S) -> Me     ! {three, self()}, receive goAhead -> S + 3 end end],
  Fun = fun(S, E) -> E(S) end,

  spawn(fun() -> Me ! choiceUpdate(Server, Fun, Choices) end),

  % all are blocked, so update while they wait
  tryUpdate(Server, fun(S) -> S + 7 end),

  % make remaining finish
  Master ! ready,
  receive {one, PID1} -> PID1 ! goAhead end,
  receive {three, PID3} -> PID3 ! goAhead end,

  % get result
  receive X2 -> ?assert(X2 =:= aborted) end,

  {ok, 49} = stop(Server).
```